VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Možnosti využití moderních herních enginů ve výuce biologie

# Possibilities of Using Modern Game Engines in the Teaching of Biology

2019 Bc. Jan Trubač

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání diplomové práce

Student: **Bc. Jan Trubač**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Možnosti využití moderních herních enginů ve výuce biologie
Possibilities of Using Modern Game Engines in the Teaching of Biology

Jazyk vypracování: čeština

Zásady pro vypracování:

Moderní herní enginy, díky svým pokročilým možnostem realtime vizualizace, dnes mají široké uplatnění napříč všemi vědními obory a nejsou již využívány pouze pro vývoj počítačových her. Využívají se při realtime vizualizacích, simulacích, nebo je lze použít pro interakci s objekty ve virtuálním prostředí. Hlavním cílem této práce je odzkoušet možnosti využití moderních herních enginů v oblasti vzdělávání, a to konkrétně v oblasti anatomie a mikrobiologie. Zaměřit se nejen na různé typy výuky, ale i na rozdílné možnosti vizualizace, a to od aplikací na desktopy a dnes např. oblíbené interaktivní tabule, tak i s použitím nových technologií jako je virtuální a rozšířená realita, které přinášejí nové možnosti.

1. Nastudujte a demonstrujte možnosti využití moderního herního enginu (Unreal Engine 4) ve výuce, kde se zaměřte na oblasti vizualizace anatomie (lidské tělo) a mikrobiologie (viry a bakterie).
2. Vytvořte demonstrační aplikaci, která bude rozdělena do třech základních částí:
- výuková část, která bude demonstrovat možnosti ve výuce zadaných oblastí.
- testovací část, určena pro ověřování znalostí.
- herní část, která bude zábavnou formou doplňovat předcházející části.
3. Výsledné aplikace upravte tak, aby je bylo možné použít jak tradičním způsobem při zobrazení na monitoru či interaktivní tabuli, tak i ve virtuální realitě.
4. Celý postup včetně vývoje aplikace pečlivě zdokumentujte, aby v práci bylo možné dále pokračovat.
5. Výsledné aplikace prakticky odzkoušejte ve spolupráci se Světem techniky Ostrava.

Seznam doporučené odborné literatury:

[1] McCaffrey, M.: Unreal Engine VR Cookbook: Developing Virtual Reality with UE4. Addison-Wesley Professional (2017). ISBN 978-01346491.
[2] Shannon, T.: Unreal Engine 4 for Design Visualization: Developing Stunning Interactive Visualizations, Animations, and Renderings (Game Design). 2017. ISBN 978-0134680705.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Martin Němec, Ph.D.**

Datum zadání: 01.09.2018

Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
*vedoucí katedry*

prof. Ing. Pavel Brandštetter, CSc.
*děkan fakulty*

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2018 .....

Poděkování patří panu Ing. Martinu Němcovi, Ph.D. za přínosné vstupy a konzultace při vzniku této práce. Dále bych rád poděkoval rodině a blízkým, kteří mě během tvorby práce a celého studia podporovali.

**Abstrakt**

Diplomová práce se zabývá možnostmi využití moderních herních enginů, a to Unreal Enginu 4 v oblasti mikrobiologie a anatomie člověka. V první části práce jsou popsány základy *Physically–based* vykreslování společně s porovnáním dvou hlavních přístupů při vytváření materiálů. V souvislosti s *PBR (Physically–based Rendering)* jsou popsány i možnosti nasvícení scén. Další kapitola je věnována technikám optimalizací, které jsou pro plynulý chod výsledných aplikací naprosto nezbytné.

Výstupy této práce zahrnují aplikace Virtuální laboratoře, krevního řečiště a anatomie člověka. Aplikace byly vytvořeny především pro zařízení virtuální reality (HTC Vive). Jedna z aplikací je rovněž vytvořena pro monitory s dotykovým ovládáním a smart tabule. Výstupní aplikace byly konzultovány se Světem techniky Ostrava, pro který jsou výsledné aplikace primárně určeny.

**Klíčová slova**: Unreal Engine, Virtuální realita, VR, Počítačová grafika, PBR, Biologie

**Abstract**

The thesis deals with the possibilities of using modern game engines, namely Unreal Engine 4, in the field of microbiology and human anatomy. The first part of the thesis describes the basics of Physically-based rendering together with the comparison of two main approaches in material creation. In the context of Physically-based Rendering (PBR) the possibilities of lighting the scenes are also described. Next chapter is devoted to optimization techniques, which are absolutely necessary to run the resulting applications smoothly.

The outputs of this work include applications of the Virtual Laboratory, Bloodstream, and Human Anatomy. Applications were created primarily for virtual reality devices (HTC Vive). One of the applications is also designed for touch screen monitors and smart boards. Output applications have been consulted with the Science and Technological Centre Ostrava for which are the applications are primarily designed.

**Key Words**: Unreal Engine, Virtual Reality, VR, Computer Graphics, PBR, Biology

# Obsah

**Přílohy**

**A Ukázka struktury části souboru pro nastavení parametrů krevního řečiště**

**B Ukázka výsledné scény virtuální laboratoře**

**C Ukázka výsledné scény krevního řečiště**

# Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| AI | – | Artificial Inteligence |
| AO | – | Ambient Occlusion |
| CPU | – | Central Processing Unit |
| DoF | – | Depth of Field |
| FPS | – | Frames Per Second |
| GPU | – | Graphics Processing Unit |
| HMD | – | Head Mounted Display |
| LOD | – | Level of Detail |
| OLED | – | Organic Light-Emitting Diode |
| PBR | – | Physically–based Rendering |
| PBS | – | Physically–based Shading |
| px | – | Pixel |
| SSAO | – | Screen Space Ambient occlusion |
| SSR | – | Screen Space Reflections |
| UE | – | Unreal Engine |
| VR | – | Virtual Reality |

# Seznam obrázků

# Seznam tabulek

# 1 Úvod

Lidé stráví studiem mnohdy více než třetinu svého života. Během této doby se každý z nás seznámil, resp. byl konfrontován, s různými metodami výuky. Existuje mnoho odborných studií, které se zabývají porovnáním metod výuky ve vztahu k jejich efektivitě. Každý z nás má jistě osobní zkušenost s tím, že si lépe pamatuje ty informace, které vnímal aktivně, se zapojením smyslů, na rozdíl od informací, které vnímal pouze z role pasivního posluchače. Současná doba se od minulosti velice liší formou výuky, přičemž jsme možná teprve na prahu toho, jak by výuka v budoucnu mohla vypadat. Počítače jsou všude kolem nás a nemusí sloužit jen k tomu, aby „počítaly“ nebo aby se na nich daly „hrát hry“. Velice významně mohou ovlivňovat náš odborný růst i množství informací, které si ze studií do života odneseme a které si budeme pamatovat. Rovněž nemůžeme zapomenout na dílo jednoho z největších pedagogů 17. století J. A. Komenského „Škola hrou“. V 21. století, dostává však Komenského myšlenka zcela novou dimenzi. Ve své práci přistupuji ke „Škole hrou“ trochu netradičním způsobem. Předvedu nástroj Unreal Engine, který byl primárně vytvořen pro tvorbu počítačových her, jako nástroj, který může zefektivnit výuku prakticky v libovolné oblasti...

Unreal Engine je v současné době jedním z nejrychleji se vyvíjejících herních enginů současnosti. Díky svému širokému poli působnosti se s ním, i když si to možná mnohdy neuvědomujeme, můžeme setkat nejen ve hrách, ale i napříč různými odvětvími – ve filmovém průmyslu, v reklamách, zpravodajství (virtuální studia, která jsou renderovaná *realtime* s využitím *Green Screen*) apod. Díky jeho možnostem a flexibilitě ho lze stejně efektivně využít i k interaktivní výuce, kde může sloužit (i díky podpoře současných zařízení pro virtuální a rozšířenou realitu) k vylepšení interakce mezi studenty a probíranou látkou.

Práce vznikala ve spolupráci se Světem techniky Ostrava [1]. Zadáním bylo vytvořit aplikace, které se zabývají tématy biologie, konkrétně mikrobiologií a anatomií. Jelikož jedním z požadavků bylo vytvořit pokud možno reálně vypadající elementy (viry a bakterie), bylo prvním krokem nastudování problematiky fyzikálního vykreslování (*Physically–based Rendering*) a s ním souvisejících konceptů při vytváření virtuálních prostředí, jako je tvorba materiálů, nasvícení scény, atd. Jelikož jsou výsledné aplikace určeny kromě smart tabulí i pro zařízení virtuální reality, bylo třeba klást důraz na techniky optimalizací vykreslování. V případě aplikací pro tento typ zařízení (konkrétně HTC Vive [2]), bylo třeba zajistit, aby scéna běžela plynule, bez záseků a na nativní frekvenci 90 Hz. Z toho důvodu je jedna samostatná kapitola věnována i optimalizacím.

# 2 Physically–based Rendering

Zjednodušeně by se dalo říci, že *PBR* je metodika vykreslování obrazu, která se snaží o dosažení co nejvěrnější reprezentace chování reálného světla ve virtuálním prostředí. Dle konkrétního kontextu se někdy můžeme rovněž setkat s pojmem *PBS*, tedy *Physically–based shading*, ten však souvisí se samotným stínováním, zatímco *PBR* popisuje vykreslování jako celek. Jedná se o často skloňovaný termín, se kterým je možné se setkat jak v *realtime*, tak v *off-line* grafice. Pochopitelně v obou těchto odvětvích grafiky bude samotný princip implementován různými způsoby. Hlavní cíle jsou však napříč všemi odvětvími stejné, a to zjednodušení a zpřehlednění *workflow* a zajištění vizuální konzistence mezi všemi objekty ve scéně za jakýchkoliv světelných podmínek. Výsledkem je tedy robustní model, který se opírá o reálné fyzikální principy a díky tomu odstraňuje nutnost odhadování jednotlivých parametrů pro vykreslování [3][4][6].

Jakožto drtivá většina současných moderních herních enginů (Unity, Frostbite, Cryengine, atd.), využívá *Physically–based* přístupu pochopitelně i Unreal Engine [7]. Pro výpočet difuzní odrazivé složky využívá Unreal Engine klasické Lambertovské funkce odrazivosti (rovnice 1):

$$f(l, v) = \frac{c_{\text{diff}}}{\pi}, \tag{1}$$

kde $c_{\text{diff}}$ je hodnota albeda pro konkrétní materiál [5].

Výpočet zrcadlové složky je v Unreal Enginu založen na Cook–Torrancově BRDF microfacet modelu (rovnice 2) [6] [7].

$$f(l, v) = \frac{D(h)F(v, h)G(l, v, h)}{4(n \cdot l)(n \cdot v)}, \tag{2}$$

kde $D$ je normální distribuční funkce (NDF) (rovnice 3).

Funkce $F$ reprezentuje Fresnelův člen. Unreal Engine zde využívá modifikované Schlickovy aproximace se sférickou Gausovou aproximací (*Spherical Gaussian approximation* (viz rovnice 4) [7].

Funkce $G$ slouží pro výpočet geometrického odrazivého útlumu (*Specular Geometric Attenuation*), který se opírá o teorii mikro–ploch (*Microfacets*) – předpoklad, že povrch je složen z malých, dokonale odrazivých ploch, kde každá plocha odráží paprsek v jednom směru dle své normály [6]. Vztahy pro výpočet funkce $G$ viz vzorec 5.

$$D(h) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2} \tag{3}$$

$$F(v, h) = F_0 + (1 - F_0) \cdot 2^{(-5.55473(v \cdot h) - 6.98316)(v \cdot h)} \tag{4}$$

$$
\begin{aligned}
k &= \frac{(Roughness + 1)^2}{8} \\
G_1(v) &= \frac{n \cdot v}{(n \cdot v)(1 - k) + k} \\
G(l, v, h) &= G_1(l) G_1(v)
\end{aligned}
\tag{5}
$$

Využití *Physically–based* přístupu se rovněž projevuje specifickým způsobem tvorby materiálů, nastavením osvětlení apod. Základní principy tvorby materiálů a způsoby nasvícení scény v UE4 jsou proto popsány v následujících podkapitolách.

## 2.1 Physically–based materiály

Materiály slouží k definování typu povrchu jednotlivých objektů scény (`Static Mesh` nebo `Skeletal Mesh`). Mezi společné vstupy pro všechny *Physically–based* materiály patří např. *Normal*, *Ambient Occlusion*, *Height*, atd. Naopak existují i vstupy, které se liší dle konkrétního *Physically–based* shaderu.

V případě využití *PBR* metodiky, existují dva přístupy pro tvorbu materiálů – *Metallic–Roughness* a *Specular–Glossines*. Oba přístupy mají své výhody i nevýhody, nicméně co se týče fyzikální přesnosti, jsou oba na stejné úrovni. Oba přístupy produkují ve finále stejný výsledek, ale dosáhnou ho různým způsobem. Unreal engine využívá první zmíněný, tedy *Metallic–Roughness* [8].

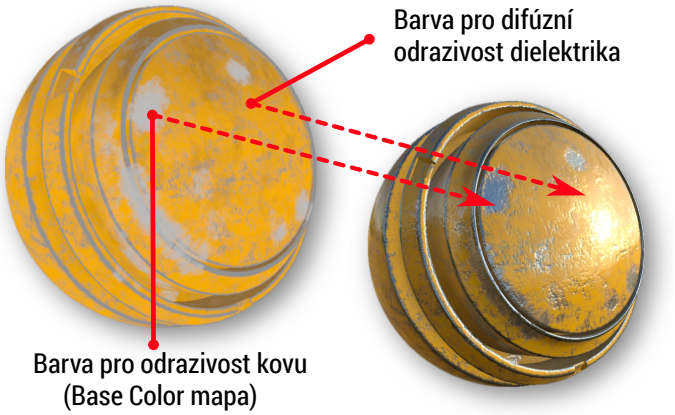
Následující část práce přibližuje základní skladbu vstupů obou metodik a rovněž porovnává jejich výhody a nevýhody.

### 2.1.1 Metallic–Roughness workflow (přístup)

Využívá tři primární vstupy do shaderu – *Base Color*, *Metallic* a *Roughness*. Výhodou tohoto přístupu, na rozdíl od *Specular–Glossines*, je, že jsou zde oba základní vstupy (*Metallic* i *Rougness*) v odstínech šedi, což přináší úsporu paměti. Za další výhodu může být považován fakt, že je tento přístup rozšířenější a lze se tak s ním setkat častěji (možnost opětovného využití textur bez nutnosti konverze do druhého systému) [9].

- **Base Color** – RGB mapa, která obsahuje barvu pro difuzní odraz u dielektrik, nebo barvu odrazu pro metalické materiály. Situaci demonstruje obrázek 1.

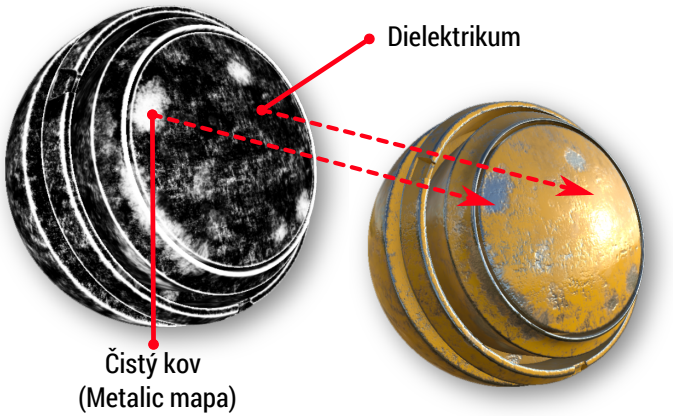
  Pro zachování korektního fyzikálního chování daného materiálu ve scéně je potřeba dodržovat správné jasové hodnoty pro difuzní odrazivost daného materiálu. Nejčastěji se jasové hodnoty pohybují v rozmezí 30–240 sRGB. Jedná se o fakt, který bývá často opomíjen a ve výsledku může právě špatný rozsah hodnot způsobit nerealistické podání celého materiálu. Například při vytváření materiálu, který má reprezentovat uhlí, by to znamenalo, že mapa (textura) na vstupu *Base Color* by neměla obsahovat hodnoty 0




Obrázek 1: Ukázka vstupu *BaseColor* u shaderu - metalické a dielektrické části materiálu

sRGB. Je to tím, že i když se uhlí může zdát jako dokonale černá hmota, stále není schopna pohlcovat veškeré světlo, tudíž musí mít sRGB hodnotu difuzní složky vyšší než nula. Korektní hodnoty pro dané materiály se pochopitelně dají změřit, či dohledat [10].

- **Metalic** – mapa ve stupních šedi, která určuje metalická a nemetalická místa (plochy) materiálu. Hodnoty 1 (255 sRGB) reprezentují 100% kov, hodnoty 0 (0 sRGB) 100% dielektrikum. Většinou má tento vstup čistě binární podobu. Úrovně šedé (např. hodnoty okolo 240 sRGB) mohou sloužit k imitaci míst, kde je kov pokryt mikroskopickou vrstvou oxidace. Obecně je však vhodné se, z důvodu zachování fyzikálních poměrů, podobným mezihodnotám vyhýbat a využívat tento vstup čistě jako černobílou masku. Ukázka efektu tohoto vstupu shaderu - viz obrázek 2.




Obrázek 2: Ukázka vstupu *Metallic* u shaderu - metalické a dielektrické části materiálu

- **Roughness** – mapa ve stupních šedi, která definuje nerovnost povrchu na mikroskopické úrovni. Hodnoty 1 (255 sRGB) reprezentují drsný povrch a hodnoty 0 (0 sRGB) potom dokonale hladký povrch. Vzhled povrchu se pak liší v rozptýlenosti odrazu.

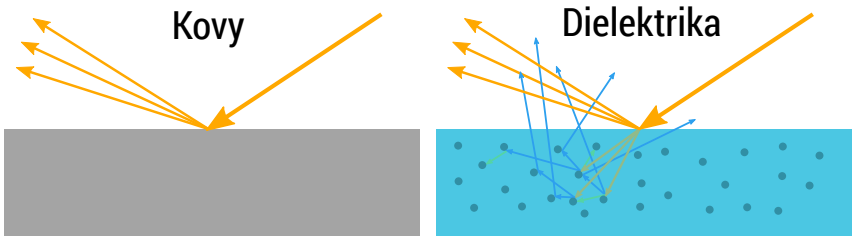
### 2.1.2 Specular-Glossines workflow (přístup)

V porovnání s *Metallic-Roughness* se jedná o starší přístup, který je však stále používán v řadě nástrojích. Primárními vstupy zde jsou *Diffuse*, *Specular* a *Glossines*.

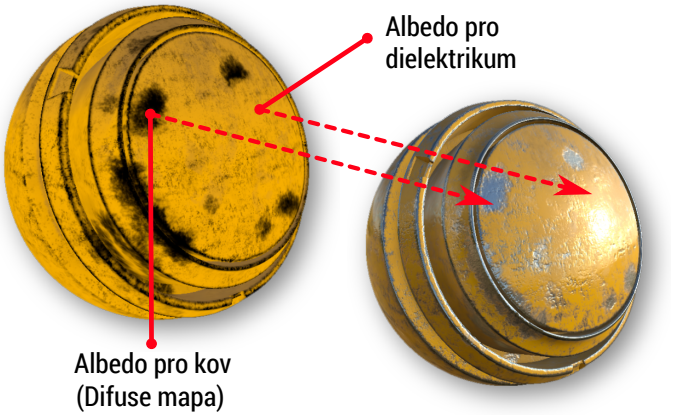
Oproti výše zmíněnému přístupu umožňuje o něco větší kontrolu nad materiálem (v podobě možnosti ovlivnění hodnoty *F0* - hodnota Fresnelovy odrazivosti pro úhel kolmý na povrch).

Nevýhodou však může být kromě větší paměťové náročnosti (*Specular* vstup je tříkanalový, tzn. RGB) také složitější proces vytváření, který vyžaduje pokročilejší znalost *Physically-based* metodiky, protože zde může dojít k porušení principu zachování energie (pokud není ošetřená uvnitř samotného shaderu). Pro splnění principu totiž nesmí být součet hodnot *Specular* a *Diffuse* větší než 1. V opačném případě by povrch emitoval více světelné energie, než by obdržel, což by bylo v rozporu s definicí zachování energie [9].

- **Diffuse (Albedo)** – RGB mapa reprezentující difúzní složku materiálů (difuzní odrazivost). Tento parametr se bude zásadně lišit v závislosti na tom, zda se jedná o materiál metalický či dielektrický. Kovy (v porovnání s dielektriky) mají absorpční koeficient řádově vyšší, a tudíž se v tomto případě podíl difúzní složky zanedbává. [11] Situaci chování světelného paprsku pro metalický a nemetalický materiál znázorňuje obrázek 3. Pro kovy je tedy difúzní složka vždy rovna nule, tudíž 0 sRGB. Většina nemetalických materiálů se nachází v oblasti 30 - 240 sRGB. Situaci ilustruje obrázek 4.

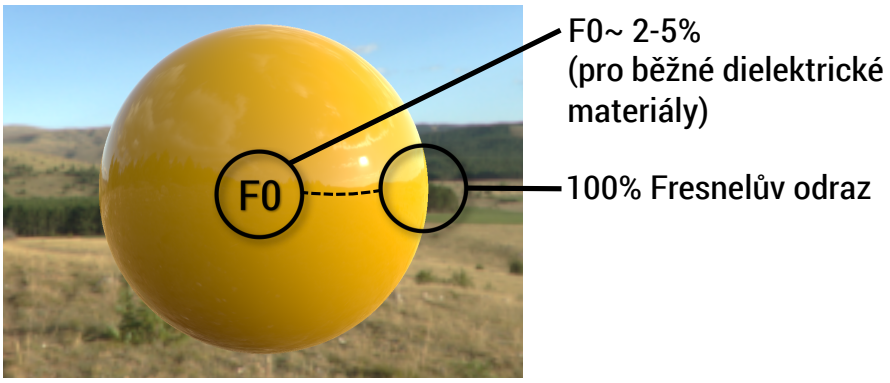



Obrázek 3: Porovnání chování světelného paprsku pro kovový a nekovový materiál

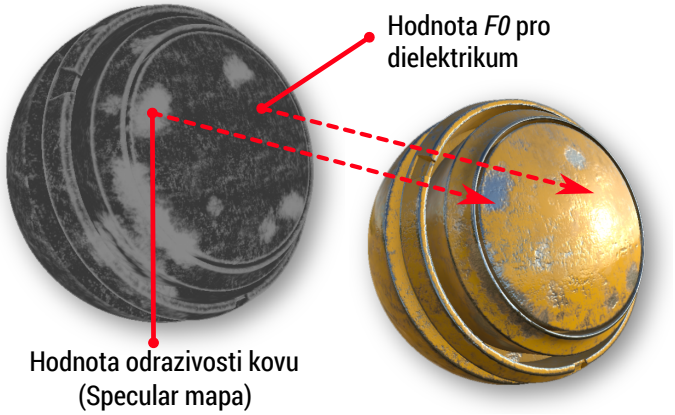



Obrázek 4: Ukázka vstupu *Diffuse (Albedo)* u shaderu - metalické a dielektrické části materiálu

- **Specular** – RGB mapa, která pro metalické materiály určuje hodnotu odrazivosti. Pro dielektrické materiály však určuje hodnotu *F0*, tedy hodnotu Fresnelovy odrazivosti pro úhel kolmý na povrch viz obrázek 5. Pro běžné dielektrické materiály je hodnota *F0* v rozmezí 2-5 %. Pro metalické materiály se hodnota pohybuje okolo 70-100 %. Ukázka tohoto vstupu materiálu ilustruje obrázek 6.



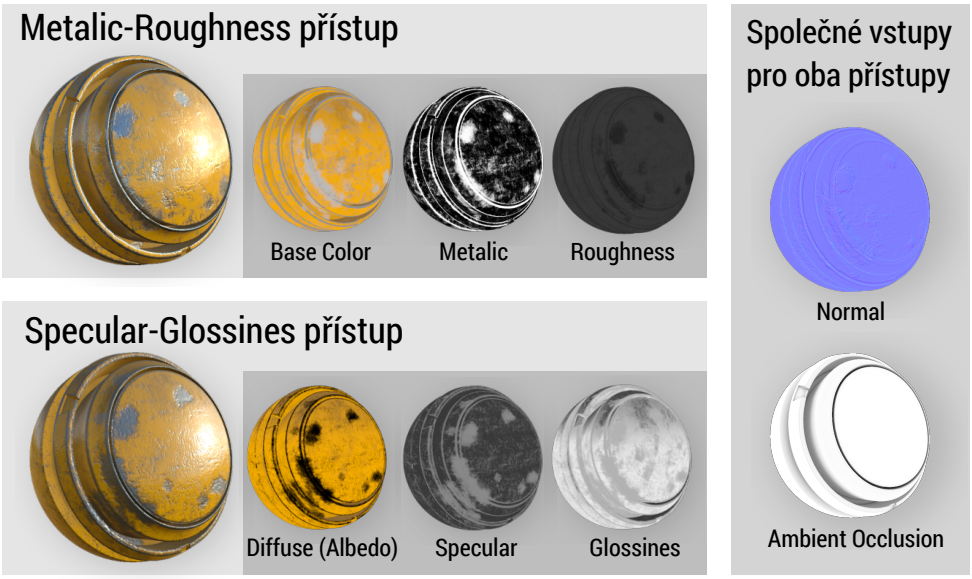

Obrázek 5: Intenzita odrazu pro úhel kolmý na povrch - *F0*




Obrázek 6: Ukázka vstupu *Specular* u shaderu - metalické a dielektrické části materiálu

- **Glossines** – mapa ve stupních šedi, která definuje mikroskopickou nerovnost daného povrchu. Jedná se prakticky o stejnou mapu jako je *Roughness* v předešlém přístupu. Jediný rozdíl je, že *Glossines* mapa je oproti *Roughness* mapě invertovaná. Černá barva zde reprezentuje mikroskopicky hrubý povrch, naopak bílá povrch dokonale hladký.

Výsledné porovnání obou těchto přístupů - *Metallic-Roughness* a *Specular-Glossines* ilustruje obrázek 7.




Obrázek 7: Porovnání *Metallic–Roughness* a *Specular–Glossines* přístupu

## 2.2 Použití shaderů v Unreal Enginu

Z těchto dvou přístupů *Physically–based* shaderů (viz obrázek 7), využívá Unreal Engine právě prvně zmíněný, tedy *Metallic-Roughness* workflow.

K tvorbě materiálů (shaderů) využívá Unreal Engine tzv. `Node-Based Graph Interface`. Jedná se o způsob tvorby shaderů, které vznikají propojováním uzlů v grafu uvnitř grafického uživatelského prostředí, kde jednotlivé uzly představují proměnné, operace nad proměnnými či funkce.

Podobného principu je využito i u samotného skriptování viz kapitola 5. Tento přístup si klade za cíl urychlení a zjednodušení celého procesu tvorby shaderů.

## 2.3 Tvorba Physically-based materiálů v nástroji Substance Designer

Substance Designer patří mezi současnou špičku mezi nástroji pro tvorbu materiálů využívanými napříč průmyslem. Jednou z hlavních předností tohoto nástroje je schopnost vytvářet plně procedurální materiály.

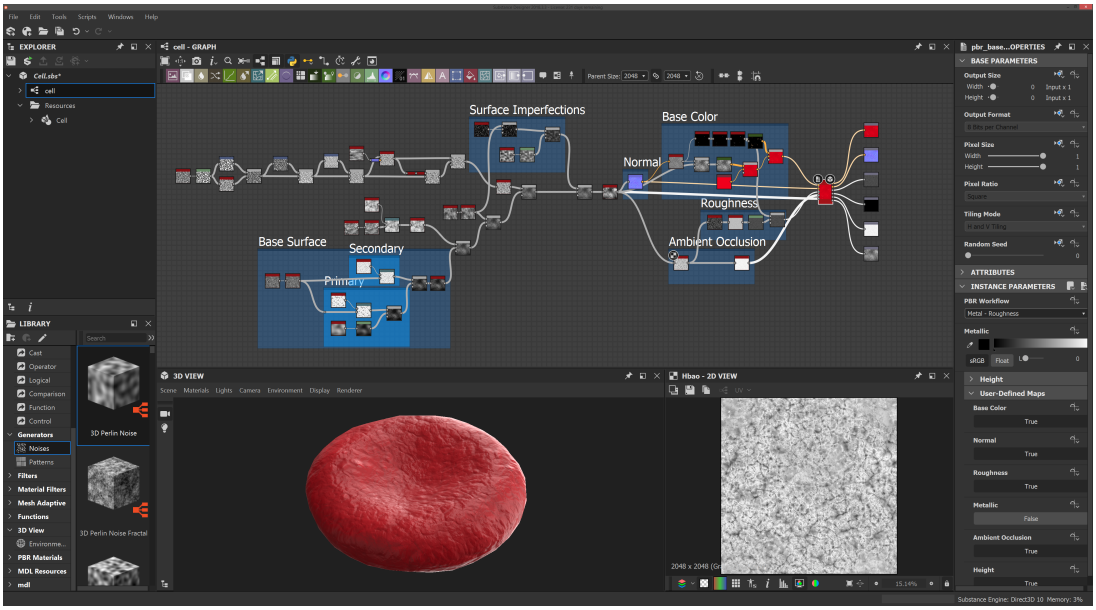
Základními stavebními prvky procedurálních materiálů jsou šumové funkce, základní geometrické tvary, vektorové obrazy, atd. Následnými úpravami a kombinacemi těchto stavebních prvků dohromady vznikají složitější struktury, které poté tvoří samotný materiál. Princip tvorby materiálu připomíná tvorbu grafu, kde dochází k propojování těchto prvků, operátorů a funkcí.

Prvním krokem při tvorbě materiálu je vytvoření výškové mapy (*Height Map*), která by měla být dostatečně detailní, neboť často slouží jako základ pro všechny ostatní mapy. Díky tomu výsledné podání materiálu většinou stojí a padá právě na tomto vstupu.

Výhodou je, že výsledné materiály jsou automaticky „bezešvé“ (*seamless*). Nevznikají tedy viditelné hrany v případě opakování stejné textury na povrchu. Další výhodou je, že po vytvoření materiálu lze měnit *seed* pro jednotlivé šumové funkce, čímž je možné vytvořit celou sadu materiálů, které budou navzájem podobné, během několika kliknutí.

Důležité parametry materiálu lze rovněž „vystavit navenek“ (*expose*). Tato možnost je užitečná např. v případě vytvoření tzv. *Substance Archive (sbsar)*, což je formát, sloužící pro přenos materiálu mezi programy, či pro publikaci materiálu obecně. Substance Designer nabízí pluginy pro řadu programů (včetně Unreal Enginu), kde je možno tento archiv naimportovat a pomocí „vystavených“ parametrů materiál upravovat. Jelikož je celý materiál procedurální, mívá tento soubor velice malou velikost (v řádech desítek kilobytů), což je další výhodou celého procedurálního přístupu. Finální textury lze poté následně vygenerovat buď z tohoto archivu, nebo pomocí funkce export přímo z nástroje Substance Designer.

Ukázka prostředí nástroje Substance Designer, při tvorbě materiálu červené krvinky viz obrázek 8.




Obrázek 8: Substance Designer - tvorba materiálu pro červené krvinky

# 3 Možnosti nasvícení scény v Unreal Enginu

Unreal Engine nabízí řadu řešení, pokud jde o možnosti nasvícení scény. Kromě základních typů světel, mezi které patří bodové-všesměrové (`Point Light`), bodové-kuželové (`Spot Light`), směrové-rovnoběžné (`Directional Light`), ambientní (`Skylight`), která se v realtime grafice používají již řadu let, se můžeme nově setkávat i s plošnými zdroji světla (`Area Lights`). Tyto plošné světelné zdroje byly do Unreal Enginu přidány společně s implementací *PBR*.

Ve spojitosti s *PBR* umožňuje současná verze enginu nastavovat hodnoty intenzity osvětlení v jednotkách lumen nebo kandela. To v praxi umožňuje přesné nastavení hodnot jednotlivých světelných zdrojů, které odpovídají jednotkám z reálného světa (světelnému toku, respektive svítivosti).

## 3.1 Základní rozdělení světel v Unreal Enginu

Kromě základních parametrů jako intenzita, barva, vyzařovací poloměr, apod., lze u každého světelného zdroje rovněž nastavit jeho mobilitu. Unreal Engine nabízí tři možnosti tohoto nastavení - `Static`, `Stationary` a `Moveable`. Nastavení `Moveable` je vhodné aplikovat pro světlo, které bude za běhu měnit své parametry nebo polohu. Na opačném konci se nachází světlo typu `Static`. Jedná se o zdroj světla, který za běhu nemůže měnit svou polohu ani parametry. Kombinací obou těchto předchozích typů je typ `Stationary`, který se sice za běhu aplikace nemůže pohybovat, ale může alespoň měnit své parametry [3].

## 3.2 UE Lightmass

Pokud víme, že se dané světlo nebude pohybovat, lze toho využít a jeho podíl nasvícení ve scéně vypočítat předem. Odborně se tento proces nazývá *Light Baking* - světlo se ve scéně spočítá přesnějšími algoritmy simulující globální osvětlení a následně se uloží do textur, nazývané `Lightmaps`, které se mapují na objekty. V terminologii UE tento proces zastřešuje pojem `UE Lightmass`.

Jelikož se pro mapování `Lightmaps` na objekty využívají UV koordináty jednotlivých objektů, je třeba, aby se jednotlivé UV plochy (UV Islands) objektů nepřekrývaly. V případě, že by objekt tuto podmínku nesplňoval, je potřeba mu vytvořit další specifický UV kanál, kde k překrývání (*UV Wrapping*) již docházet nebude a bude tak možné ho použít k mapování `Lightmaps`. Alternativní metodou je automatické vygenerování `Lightmap UVs` při importu 3D modelu do Unreal Enginu. Tuto metodu lze doporučit v případě jednoduchých modelů s jednoduchou topologií. V případě tvarově složitějších a detailnějších modelů je vhodné `Lightmap UVs` vytvořit ručně.

Každému objektu ve scéně lze následně stanovit rozlišení pro generování `Lightmaps`. Platí, že velikost objektu ve scéně je přímo úměrná rozlišení jeho `Lightmap`. Pro usnadnění nastavování adekvátního rozlišení je v enginu možnost vizualizace tzv. `Lightmap Density`.

Jedná se o užitečnou vlastnost, která eliminuje nutnost odhadování velikosti rozlišení. Z důvodu úspory paměti je vhodné se držet velikostí rozlišení $2^x$.

Pro výpočet `Lightmap` bylo ve výsledných aplikacích této práce využito *Third-Party* řešení, a to v podobě GPU Lightmap Baker - *Luoshuang's GPULightmass* [12]. Jedná se o nástroj, jehož části se nakopírují přímo do složky s enginem. Samotné použití je pak naprosto totožné jako v případě klasického `UE Lightmass`. Hlavní výhodou toho nástroje je značné snížení doby výpočtu `Lightmaps`.

### 3.2.1 UE Lightmass Portals

V případě využití venkovního světla, jako primárního osvětlovacího zdroje pro interiérovou scénu, bývá často problém s kvalitou vygenerovaných `Lightmaps`. V zákoutích a místech, kde nedopadá přímé světlo, se tak můžeme setkat s artefakty (skvrnami) a celkovým nepřesným podáním vnitřního nasvícení.

Řešením je přidání tzv. `Lightmass Portals` do otvorů, kudy do místnosti vniká venkovní světlo. Tyto portály slouží k usměrnění paprsků vnikajících do interiéru. Díky tomu je do místnosti soustředěno více paprsků a výsledkem je přesnější reprezentace odraženého světla v místnosti. [13].

# 4 Způsoby optimalizace

Obecným pojmem optimalizace se v 3D počítačové grafice označuje proces, který má za cíl snížit časovou, nebo paměťovou náročnost pro vykreslení snímku. Tyto optimalizace je třeba zavést, aby bylo možno dosáhnout např. věrnějších (fotorealistických) výsledků, většího počtu objektů ve scéně, na pohled detailnějších objektů atd. I když to může vypadat, že je tento výčet spíš opakem optimalizace, nemusí tomu tak vždy být. Mnohdy optimalizace neznamenají pouhé vypnutí toho či onoho efektu, či snížení kvality celkového obrazu. Ideálním řešením je scénu optimalizovat tak, aby byl výsledný vzhled zachován, ale i přesto došlo k navýšení snímkovací frekvence. Mnohdy lze toho dosáhnout pouze vhodným návrhem celé aplikace a dodržováním stanovených limitů, které se mohou týkat například počtu vykreslovaných trojúhelníků, množství dynamických prvků ve scéně, počtu světel atd.

Správný přístup by měl spočívat v kontinuálním přehledu o náročnosti scény během samotného vývoje. Ve finále je mnohem lepší volbou vytvářet scénu s ohledem na předem stanovené limity (maximální počet vykreslovaných trojúhelníků, maximální počet materiálů na objekt, způsob nasvícení atd.), než se optimalizacemi zabývat v poslední části vývoje, kdy bude daleko těžší modifikovat její části, díky celkově větší komplexitě scény [14].

Optimalizace jsou ještě důležitější v případě vývoje aplikací pro virtuální realitu. Díky kombinaci vysoké snímkovací frekvence (u zařízení HTC Vive, 90 Hz), relativně velkého rozlišení OLED panelů v headsetu a nutnosti renderovat celou scénu prakticky dvakrát (stereo), hrají optimalizace klíčovou roli. Jelikož je několik výsledných aplikací této práce zaměřeno pro zařízení virtuální reality HTC Vive, byl problematice optimalizačních technik věnován patřičný prostor. Klíčové poznatky získané z procesů optimalizací aplikací v UE (včetně poznatků z výsledných aplikací této práce) jsou shrnuty v následujících podkapitolách.

## 4.1 Rozlišení

Jedním z hlavních faktorů, který se na rychlosti vykreslování podepisuje, je rozlišení vykreslovaného obrazu. Současný trend se posouvá směrem k 4K rozlišení (3840x2160). V porovnání s FullHD rozlišením (1920x1080) se jedná o čtyřnásobný počet pixelů, které je nutno zpracovat a zobrazit. V moderních hrách, kde se provádí řada operací nad každým pixelem, se jedná o extrémní zátěž pro grafickou kartu. Z těchto důvodů vznikají nové techniky, které využívají neuronových sítí a strojového učení k modifikaci vykresleného obrazu s nižším rozlišením na obraz ve vyšším rozlišení. [15]

## 4.2 3D geometrie

Se stále vyvíjejícími se grafickými kartami se zvyšuje i počet trojúhelníků, které je možno vykreslit. Na druhou stranu roste i složitost scén, které jsou daleko detailnější a povrchy využívají mnohdy i tzv. tesselace. [1] Právě díky teselaci a vysokému počtu zhuštěných trojúhelníků se setkáváme s tzv. *Quad overshading*, nebo *Quad overdraw* problémem (blíže je tento problém popspán v podkapitole 4.2.2), který se velkou měrou podílí na rychlosti vykreslování [16].

Co se týče počtu trojúhelníků obecně, tak optimálním řešením je stanovit si na začátku tvorby scény maximální limit polygonů (dle možnosti platformy, pro kterou je daná aplikace vyvíjena) a tohoto limitu se pak následně držet. Rovněž je vhodné stanovit si pravidla, jak detailní mají být různé typy objektů vůči sobě. Například v případě RPG hry je hráčova postava nejčastěji vykreslovaným objektem ve scéně. Proto lze vytvořit detailnější postavu, na úkor objektů v okolním prostředí, které nehrají tak důležitou roli.

### 4.2.1 LOD 3D objektů

Při použití detailních 3D objektů se počet vykreslovaných trojúhelníků snadno dostane na hranici, který již není současný HW schopen v požadovaném čase zpracovávat. Extrémním příkladem může být situace, kde je vykreslovaný objekt (např. strom), vzdálen několik kilometrů od kamery (hráče), takže ve výsledném obraze zabírá strom plochu pouze několika jednotek pixelů. V takovém případě je zbytečné plýtvat výkon na detaily, které nejsou vidět, a tudíž se nabízí řešení přepínat mezi různě detailními verzemi daného objektu dle jeho vzdálenosti od kamery. Této technice se říká *Level of Detail*. Pomocí ní lze efektivně snížit počet vykreslovaných polygonů při zachování vizuální kvality.

V praxi to pak vypadá tak, že je nutno každý objekt vytvářet v několika verzích s degradujícím počtem trojúhelníků. Unreal Engine však v tomhle ohledu nabízí řešení v podobě automatického generátoru `LOD` objektů. Díky tomu stačí vytvořit pouze jednu (detailní) verzi a verze s nižším počtem polygonů si nechat automaticky vygenerovat enginem [17].

### 4.2.2 Quad Overshading/Overdraw problém

Kromě snížení samotného počtu trojúhelníků poskytuje technika *LOD* objektů řešení ještě jednoho problému - *Quad Overdraw*. Jedná se o poměrně málo známý problém, který má však

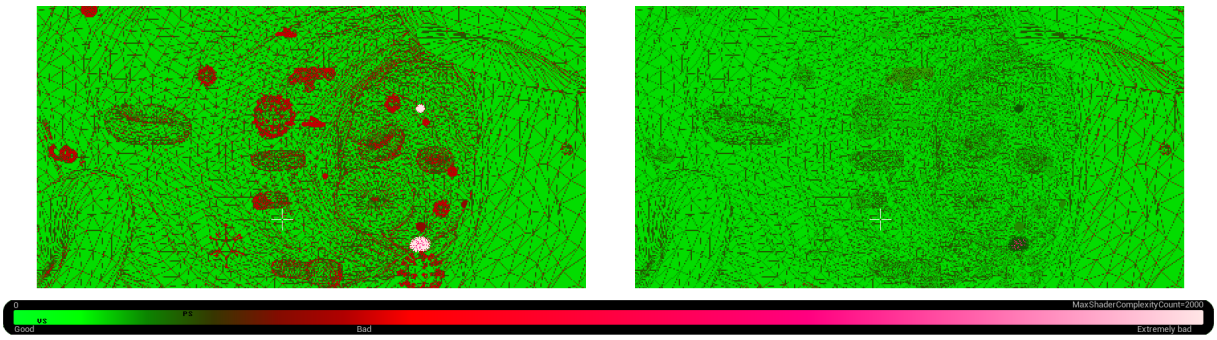
[1] Tesselace je způsob zjemňování topologie objektu (meshe). Většinou se využívá v kombinaci s Displacement mapou k zvýšení detailnosti povrchu objektu na úrovni samotných trojúhelníků. Unreal Engine nabízí dvě možnosti tesselace - Flat a PN Triangles. Flat pouze rozdělí trojúhelníky, zatímco PN Triangles rovněž vyhladí (zaoblí) povrch meshe.

Jelikož se jedná o poměrně náročný efekt, je úroveň teselace většinou spojená se vzdáleností daného objektu od kamery. Čím blíže je tedy objekt ke kameře tím „hustší“ je jeho geometrická síť. V současnosti se s tímto efektem můžeme setkat i u terénů, kde je využito detailní topologie např. k dynamickému vkládání stop v blátě, vytváření kráterů po výbuchu granátu či podobných efektů. Díky fyzické změně topologie z tohoto efektu těží i Post Process efekty (jako třeba AO), které tak mohou využít detailní topologii pro přesnější reprezentaci daného efektu.

velké dopady na rychlost vykreslování.

Problém vzniká u trojúhelníků, které jsou velice malé či úzké. Typicky problém nastává u vzdálenějších objektů, nebo objektů, které jsou na horizontu, kde jsou vlivem perspektivy trojúhelníky již hodně podlouhlé. Stínování trojúhelníků se totiž neprovádí na jednotlivých pixelech, ale na větších blocích (u současných moderních grafických karet po blocích o velikostech 2×2 vepx). Díky tomu je ve finále u malých trojúhelníků řada pixelů „zahozena" a jejich výpočet je tedy zbytečný. Řešením problému je využití techniky `LOD`. Problém eliminuje tím, že s rostoucí vzdáleností objektu od kamery používá méně detailní verzi objektu - s nižším počtem trojúhelníků, které jsou díky tomu i větší, a tudíž nedochází k tomuto problému [16] [18].

Situaci ilustruje obrázek 9 z aplikace krevního řečiště. Zelená barva značí malý, až nulový *Quad Overshading*. Hodnoty od červené do bílé značí vyšší až extrémní hodnoty. Z obrázku je patrné, že s využitím techniky *LOD* na objektech elementů (obrázek 9 vpravo) se problém *Quad Overshading* značně eliminoval




Obrázek 9: Vizualizace *Quad Overshading* v enginu. Využitím techniky *LOD* došlo k eleminaci *Quad Overshading* problému (obrázek vpravo)

### 4.2.3 Paměťové úspory v souvislosti s 3D objekty

Co se týče paměťové náročnosti ve spojitosti s 3D objekty, je ideální využívat co nejmenší počet *Smoothing Groups* [19]. Protože pokud daný vrchol leží na pomezí dvou či více *Smoothing Groups*, musí být poté tento stejný vrchol v paměti reprezentován několikrát, pro každou skupinu. Pokud to je tedy možné, je vhodné využít pouze jedné skupiny na celý objekt.

Podobná pravidla platí i u procesu *UV Unwrapping*. Je opět vhodné rozdělovat mesh co možná nejméně tzn. vytvářet co nejmenší počet tzv. *UV Islands*. Samozřejmě se ve finále nemusí jednat o nějaký velký paměťový dopad, nicméně i tak je stále dobré na to při vytváření modelů brát ohled [20].

## 4.3 Materiály

S materiály souvisí mnoho faktorů, které se značnou měrou podílejí na celkové náročnosti scény. Při tvorbě objektů je důležité zvážit, kolik materiálů bude daný model používat. S každým dalším přidaným materiálem na objekt totiž přibude také další *Draw Call*. [2] Z toho důvodu je dobré počtem jednotlivých materiálů na objektu „šetřit“ a pokud to je možné, zbytečně objekty materiálově neštěpit.

Samozřejmě existují případy, kdy si nelze vystačit pouze s jedním materiálem. Například se může jednat o situace, kdy se objekt skládá z neprůhledných, průsvitných, či průhledných částí. V takovém případě je nutno použít dvou různých materiálů, protože průsvitné či průhledné materiály se vykreslují jiným způsobem než neprůhledné.

Typickým případem, kdy je nutno použít více materiálů může být strom. V takovém případě je třeba vytvořit minimálně dva materiály. Jeden `Opaque`, který je neprůhledný a bude tak aplikovaný na kmen a hrubší větve a druhý, `Masked` materiál (využívající černobílou masku), který bude použit na listy. Předpokladem je samozřejmě fakt, že listy na stromu jsou vymodelovány jako obdélníky, kde se až v shaderu stanoví, která část má být průhledná a která nikoliv.

### 4.3.1 Overdraw u průhledných či průsvitných materiálů

Podobně, jako tomu bylo v případě meshů, se i zde setkáváme s pojmem *Overdraw*. Zde se jedná o několikanásobné překreslení daného pixelu v jednom vykresleném snímku. Problém vzniká v případě, kdy se přes sebe vykreslují jednotlivé průhledné plochy polygonů. Na rozdíl od neprůhledného objektu, kde se objekty v zákrytu díky *Occlussion Culling* [22] ani nevykreslují, je u průhledného třeba vykreslit i tyto objekty v zákrytu. V případě více průhledných objektů za sebou musí nejprve dojít k seřazení [3] dle jejich vzdálenosti od kamery. Až poté je možné je postupně dle vzdáleností vykreslit. Díky tomu je vykreslování objektů využívajících materiály s průhledností komplikovanější a tedy i časově náročnější.

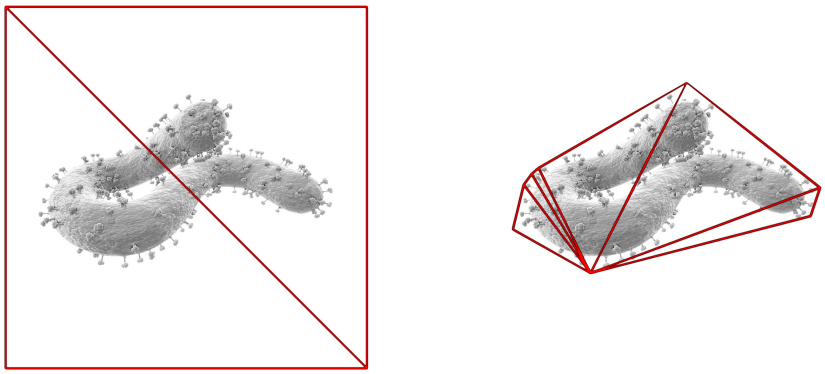
Optimalizovat *Overdraw* lze několika způsoby. Nejefektivnějším způsobem je vytvoření vhodného meshe, který bude co nejblíže obepínat neprůhlednou část textury a tím bude eliminovat velikost plochy, která bude ve finále průhledná, viz obrázek 10. Díky tomu sice dojde ke zvýšení počtu polygonů daného objektu, ale dojde rovněž k značnému urychlení, protože nebude nutné několikrát překreslovat tak velké průhledné plochy objektů v obraze. Problém *Overdraw* je totiž ze své podstaty větší než vykreslení několika trojúhelníků navíc.

Obrázek 10 demonstruje optimalizaci meshe pro částicový efekt. Původní, čtvercový mesh, byl pomocí funkce `Particle Cutout` automaticky upraven, aby eleminioval velké prázdné

[2] Draw Call obsahuje veškeré informace o shaderech, texturách, objektech, bufferech, atd. Tato data připraví procesor (CPU) a následně je přes command buffer odešle grafické kartě (GPU), která je zpracuje a vykreslí finální obraz [21].

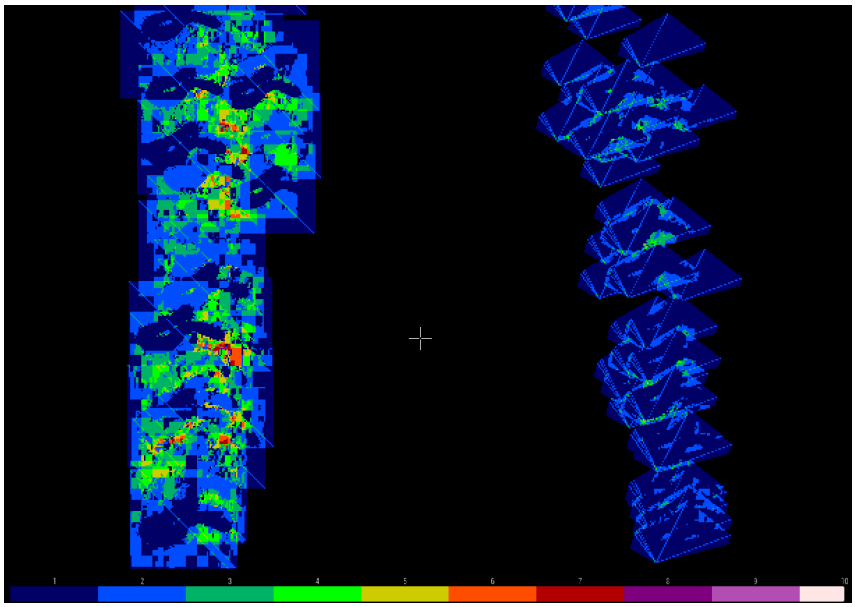
[3] Unreal Engine třídí vzdálenosti pouze samotných objektů. Polygony uvnitř objektu jsou vykresleny v takovém pořadí, v jakém jsou zapsány jednotlivé vrcholy do tzv. Meshes Index Buffer [23]

plochy. Tato funkce je součástí částicového editoru v UE, což je obrovskou výhodou pro tvorbu optimalizovaných částicových efektů, minimalizující *Overdraw*.



Obrázek 10: Optimalizace meshe pomocí funkce *Particle Cutout* (optimalizovaná verze vpravo)

Porovnání využití původního a modifikovaného částicového efektu využívající funkce `Particle Cutout` viz obrázek 11. Z tohoto obrázku je patrné, že v částicovém efektu napravo je daleko méně zelených a červených ploch, z čehož vyplývá, že částicový efekt napravo je méně zatěžující [24].



Obrázek 11: Porovnání náročnosti původního, čtvercového meshe (vlevo) s optimalizovaným (vpravo) z pohledu *Quad Overdraw*

Tento způsob optimalizace je nejvíce efektivní v případě, kdy jsou vykreslované polygony blízko kamery a průhledné plochy polygonů jsou tedy největší. S rostoucí vzdáleností těchto polygonů od kamery již však není tento způsob optimalizace tak efektivní, protože průhledné plochy jsou malé a přidaná geometrie by již více zatěžovala vykreslování. Ideálním řešením je

využití zmíněné techniky `LOD` objektů, kdy stačí pro vzdálené objekty použít jednodušší verzi meshe, s menším počtem polygonů.

*Overdraw*, potažmo náročnost vykreslování jednotlivých pixelů, lze v Unreal Enginu vizualizovat v reálném čase pomocí volby zobrazení `Shader Complexity`.

### 4.3.2 Další metody optimalizace materiálů

Jelikož Unreal Engine umožňuje nastavit materiál zvlášť pro každý `LOD`, je možné vytvořit materiály, které budou na vzdálenějších objektech využívat jednodušší materiály s menším počtem instrukcí, nebo budou dokonce vynechávat některé vstupy materiálu. V extrémním případě lze pro velmi vzdálené objekty využít pouze kanál s albedo texturou (Base Color).

Pro další snížení paměťové náročnosti u materiálů je možné použít tzv. `RGB Mask Packing`. Jedná se o formu ukládání jednotlivých černobílých map (Metalic, Roughness, Ambient Occlusion, atd.) do jednotlivých RGB kanálů jediné textury. Díky tomu operuje GPU pouze s jednou texturou, na místo několika jednotlivých. Díky této optimalizaci může rovněž finální aplikace zabírat méně místa na disku, protože textury patří mezi největší soubory, s kterými současné hry či podobné aplikace operují. Jedná se tedy i o způsob, jak znatelně snížit velikost celé výsledné aplikace.

Menší počet textur rovněž „šetří“ tzv. `Texture Streaming Cache`. Jedná se o *cache*, kde si Unreal Engine uchovává textury v různých rozlišeních. `Texture Streaming` se dá zjednodušeně považovat za jakéhosi tvůrce `LOD` pro textury [3]. Velký počet textur však může plně zaplnit tuto *cache*, což se ve výsledku projeví pomalým načítáním textur v požadovaném rozlišení.

## 4.4 Světla

Důležitými prvky scény (z pohledu náročnosti) jsou světelné zdroje. Ty se dle zvoleného typu mohou značnou měrou podílet na plynulosti vykreslování. V případě nasvícení scény statickými (`Static`) světly je nasvícení uloženo do `Lightmaps`, a tak je ve většině případů dopad těchto světel za běhu prakticky nulový. Využití těchto světel je tedy ideální volbou v případě vytváření scén pro virtuální realitu, kde je nutné dbát ohled na rychle a plynulé vykreslování.

V případě použití pohyblivých světel (`Moveable`), které za běhu mohou měnit jak svou pozici, tak i parametry, je nutné dbát na jejich efektivním používání. Jelikož se jedná a o nejnáročnější typ, je možné nesprávným použitím těchto světel scénu přetížit.

Snížit náročnost jednotlivých světel lze i pomocí vypnutí metody `Inverse Square Fallof` a na místo ní využít základní verzi přechodu, která sice není fyzikálně korektní, ale v určitých situacích může generovat podobný výsledek i s menším poloměrem vyzařování - čímž může dojít k ušetření výkonu.

Kompromisem mezi statickými a pohyblivými světly je světlo typu `Stationary`, které ale přináší také své limitace v podobě maximálního počtu světel, které se mohou navzájem překrývat.

### 4.4.1 Lightmaps

`Lightmaps` jsou efektivním řešením jak snížit náročnost světel, u kterých je předem známo, že se nebudou pohybovat, či jinak měnit své parametry za běhu. Výhodou je rovněž přesnější nasvícení, kterého je dosaženo díky složitějším algoritmům, které jsou použity při jejich generování. Nevýhodou však může být jejich delší výpočetní čas (pro vyšší kvalitu `Lightmap` ve vyšším rozlišení, může být čas výpočtu i v rámci desítek hodin). Další nevýhodou může být paměťová náročnost. Důležité je správně nastavit rozlišení `Lightmap` u jednotlivých objektů, dle jejich fyzické velikosti ve scéně. Příliš velké rozlišení textur má za následek velkou paměťovou náročnost a to nejen pro VRAM ale rovněž pro disk (`Lightmaps` mohou na disku zabírat i stovky MB).

### 4.4.2 Stíny

Stíny patří mezi velice náročné prvky scény. Vykreslení dynamických stínů z jednoho světelného zdroje může, v některých případech (např. při velkém rozlišení stínu), trvat i několik milisekund. Z tohoto důvodu, pokud je to možné, lze doporučit vykreslování stínů pro dané světlo vypnout. Vykreslování stínů je rovněž možné nastavit pro jednotlivé objekty. Pokud tedy není nutné, aby objekt vrhal stín, je vhodné stín vypnout.

Dalším důležitým parametrem, který lze snadno přehlédnout, je vyzařovací poloměr světla. I zde je snahou držet poloměr co nejmenší, protože může ve výsledku ušetřit dobu nutnou pro vykreslení stínů (které mohou být relativně daleko, tudíž méně viditelné), ale rovněž i čas potřebný pro vykreslení daného světla v pixel shaderu. U stínů je také možno snížit kvalitu a to pomocí příkazu `sg.ShadwoQuality` [20].

## 4.5 Post Process efekty

`Post Process` efekty se mohou rovněž značnou měrou podílet na rychlosti vykreslování snímku. Mezi nejnáročnější patří například *Anti-aliasing* (*AA*), *Screen Space Reflections* (*SSR*), *Ambient Occlusion* (*AO*), *Depth of Field* (*DoF*) či *Bloom*. Jednotlivé efekty mají možnost nastavení úrovně kvality, tzn. že se opět jedná o nalezení určitého kompromisu mezi požadovaným vzhledem a rychlosti vykreslování. Je důležité rovněž zmínit, že všechny tyto efekty jsou přímo závislé na velikosti rozlišení vykreslovaného obrazu, nikoliv například na počtu trojúhelníků ve scéně.

## 4.6 Analýza náročnosti vykreslování, zjišťování zdrojů problémů

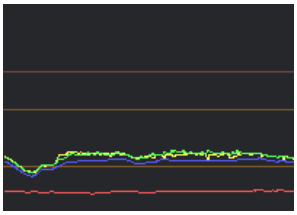
Unreal Engine disponuje širokou paletou nástrojů, které umožňují provést podrobnou analýzu zatěžujících elementů, a díky tomu přesně určit elementy, které je potřeba optimalizovat.

Pro základní monitorování běhu vykreslování postačí příkaz `stat FPS` a `stat UNIT`. Tyto příkazy vykreslí do okna informace o snímkovací frekvenci respektive informace o jednotlivých

časech výpočtu pro `Game Thread`, `Render Thread (Draw)` a `GPU`, viz obrázek 12. Tento jednoduchý příkaz umožňuje lokalizovat, zda je úzké místo na CPU, nebo na GPU. V tomto konkrétním případě je problém na straně CPU, konkrétně `Render Thread (Draw)`, i když je u GPU uveden vyšší čas. Je to z toho důvodu že výpočet na GPU mohl začít až po dokončení `Render Thread (Draw)`. GPU tedy byla v tzv. *Stall* stavu (stav čekání na data) [3][20].

Pro informaci o snímkovací frekvenci v čase slouží příkaz `stat UnitGraph`. Ukázka výpisu tohoto příkazu viz obrázek 13.


Obrázek 12: Ukázka výpisů příkazů *stat FPS* a *stat UNIT*



Obrázek 13: Ukázka grafu, který vykreslí příkaz *stat UnitGraph*

Obecně je vhodné mít informace o rychlosti vykreslování stále zapnuté, neboť je daleko snazší si všimnout náhlého poklesu snímkovací frekvence při tvorbě scény, než hledat chybu v už hotové scéně.

Pro přesné hodnoty snímkovací frekvence je rovněž vhodné v nastavení projektu vypnout funkci `Smooth Frame Rate`. Tato funkce eliminuje prudké výkyvy (*spikes*), které by mohly zkreslovat výsledky měření.

### 4.6.1 Game Thread

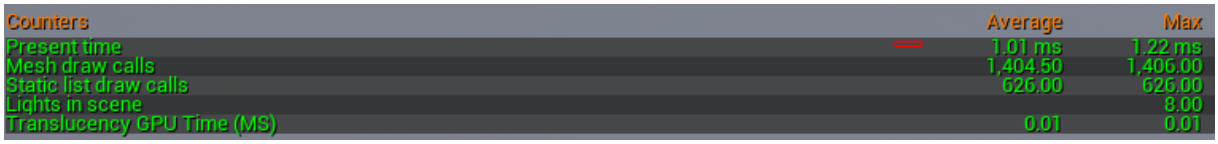
Pokud dojdeme k závěru, že nejvíce času zabírá výpočet v `Game Thread`, znamená to, že nejvíce času zabere výpočet logiky hry. Jelikož většinou nebývá `Game Thread` v porovnání s ostatními prominentní, bývá většinou problém v podobě např. chyby v blueprintu, kde se může provádět příliš mnoho operací na snímek (`Game Tick`). Případně může být problém s velkým počtem tzv. `RayCast`, komplikovanou fyzikou, složitou *AI*, apod.

### 4.6.2 Render Thread

V případě, že se čas u `Render Thread` blíží času pro vykreslení celého snímku (`Frame`), je patrné, že vykreslování zpomaluje CPU. Nejčastější příčinou v tomto případě bývá příliš komplexní scéna s velkým počtem *Draw Calls*, kdy procesor „nestíhá“ grafické kartě připravovat data pro vykreslování. Řešením může být již zmiňované snížení celkového počtu objektů a materiálů, či využití tzv. `Instanced Mesh`, díky kterému mohou být všechny instance objektů zpracovány přes jediný *Draw Call* [25]. Tohoto přístupu využívá Unreal Engine například automaticky v případě využití `Foliage Tool`. Pro zjištění celkového počtu

*Draw Calls* slouží příkaz `stat SceneRendering`. Ukázka výstupu tohoto příkazu viz obrázek 14.

Další možnou zátěží pro `Render Thead` může být například `Occlusion Culling`. V některých případech je dokonce výhodnější `Occlusion Culling` vypnout.




Obrázek 14: Ukázka výpisu příkazu stat SceneRendering

### 4.6.3 GPU

Pokud je hodnota `GPU` (v zobrazeném výpisu) poměrově větší než `Render Thread`, nebo `Game Thread`, je nejpravděpodobnější, že úzkým článkem v řetězci je právě grafická karta. Mezi typické problémy, které zatěžují GPU, patří např. vykreslování dynamických stínů, velký počet trojúhelníků (spojený např. s tesselací), Post Process efekty atd. Ve spojitosti s GPU je možno rovněž narazit na limity její paměti (*VRAM*), způsobeny např. velkým rozlišením vykreslovaného obrazu, velkým rozlišením nekomprimovaných textur atd.

Pro identifikaci konkrétních problémů ve spojitosti s GPU slouží příkazy `stat GPU` nebo detailnější `profile GPU`. Pro zobrazení počtu vykreslovaných trojúhelníků lze použít příkaz `stat RHI`.

Základní příkazy, pro analýzu náročnosti, jsou pro přehlednost vyneseny do tabulky 1.

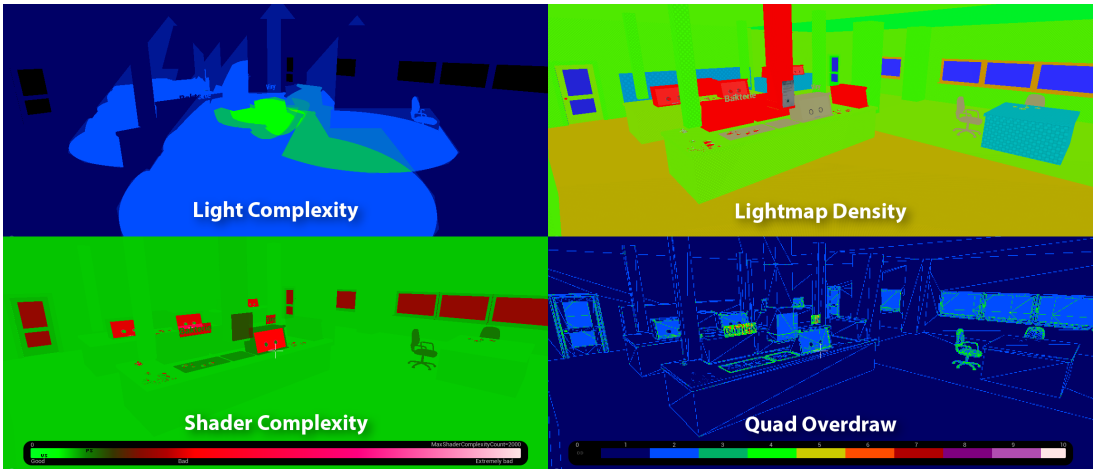
Tabulka 1: Tabulka konzolových příkazů pro analýzu náročnosti vykreslování

| Příkaz | Akce |
|---|---|
| stat FPS | zobrazení snímkovací frekvence |
| stat UNIT | zobrazení časů Game, Draw Thread a GPU |
| stat UnitGraph | zobrazení grafu pro jednotlivé časy vykreslování |
| stat SceneRendering | zobrazí obecné statistiky vykreslování (včetně počtu Draw Calls) |
| Profile GPU | zobrazí okno s hodnotami jednotlivých průchodů (passes) pro GPU |
| stat GPU | zobrazí základní informace o vykreslování průchodů do viewportu |
| stat RHI | zobrazí informace o využití paměti pro textury a počet trojúhelníků |
| stat startfile/stat stopfile | zahájení a ukončení záznamu pro pozdější analýzu |
| stat Memory | zobrazí informace o využití RAM včetně Texture Streaming pool |
| stat none | odstraní všechny výpisy z obrazovky |
| r.ScreenPercentage | nastavení požadované velikosti rozlišení (v procentech) |
| FreezeRendering | „zmrazí“ vykreslování, vhodné např. pro kontrolu Occlusion Culling |

### 4.6.4 UE Optimization Viewmodes

Pro reálnou vizualizaci možných problémů uvnitř samotného viewportu slouží tzv. `Optimization Viewmodes`. Jedná se o speciální zobrazení, která barevně vizualizují náročnost pro konkrétní problém např. překrývání světel (`Light Complexity`), `Quad Overdraw`, `Lightmap Density`, `Shader Complexity` atd. Souhrnné zobrazení těchto několika režimů ilustruje obrázek 15.

Z obrázku 15 vyplývá, že vyobrazená scéna je poměrně nenáročná. Z pohledu `Light Complexity` dochází pouze k minimálnímu překrytí dynamických světel. Co se týče `Lightmaps Density`, je možné u červeně zbarvených objektů snížit rozlišení `Lightmap`. Na druhou stranu, díky malému počtu objektů, se snížení tohoto rozlišení na výkonu pravděpodobně nepodepíše. Z pohledu `Shader Complexity` reprezentuje scéna v podstatě ideální stav. Jedinými náročnějšími shadery ve scéně jsou shadery pro skla. Ty však již, díky svému způsobu vykreslování a s tím souvisejícími operacemi, prakticky nelze optimalizovat. Co se týče `Quad Overdraw`, tak zde rovněž nedochází k problémům, poněvadž většina trojúhelníků na obrazovce je dostatečně velká.




Obrázek 15: *Optimization Viewmodes* demonstrovány na scéně laboratoře

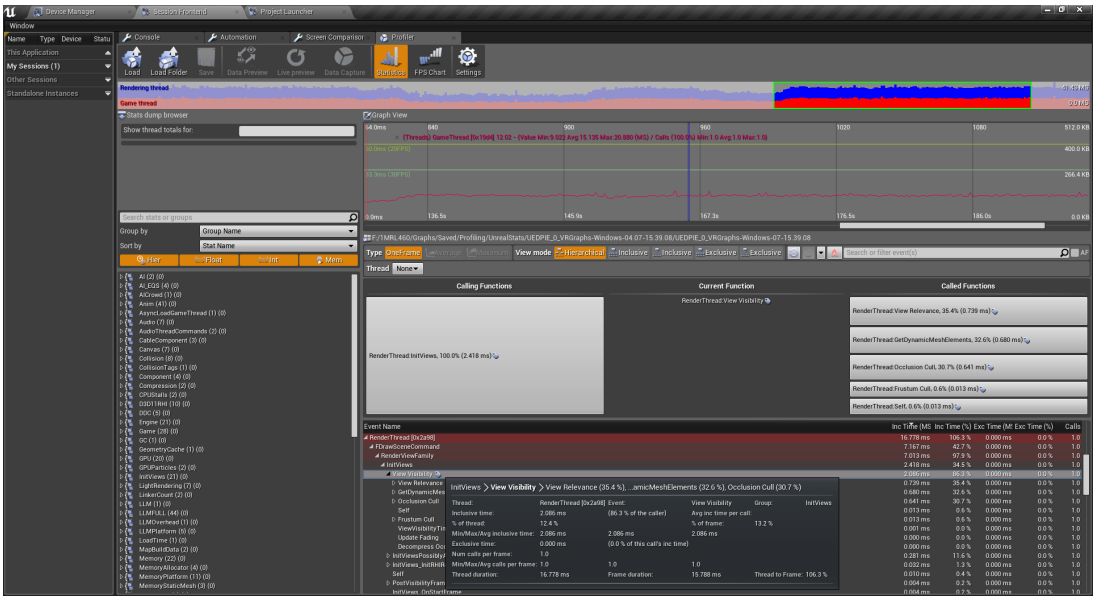
### 4.6.5 UE FrontEnd

Kromě již zmíněných příkazů, které vypisují některé základní údaje vykreslování, je možno využít i pokročilejšího nástroje, kterým je `Unreal Frontend`. Tento nástroj slouží kromě řady jiných funkcí i k podrobné analýze náročnosti jednotlivých prvků (Profiling) [3].

Tento nástroj pracuje v *off-line* režimu, tedy nemonitoruje dění v reálném čase. Dokáže však otevřít záznamy, které je možno za běhu aplikace vytvořit pomocí příkazů `stat startfile` a `stat stopfile`. Záznam je vhodné nechat spuštěn alespoň několik sekund.

Po vytvoření záznamu stačí v `Unreal Frontend` otevřít vygenerovaný soubor s daty, naleznout v seznamu patřičnou záložku (většinou `Game Thread` nebo `Render Thread`) a

„dolistovat“ se na zdroj problému. Na obrázku 16 lze vidět uživatelské rozhraní `Unreal Frontend` s načtenými daty.



Obrázek 16: Ukázka uživatelského rozhraní nástroje UE Frontend

### 4.6.6 Intel Graphics Performance Analyzers

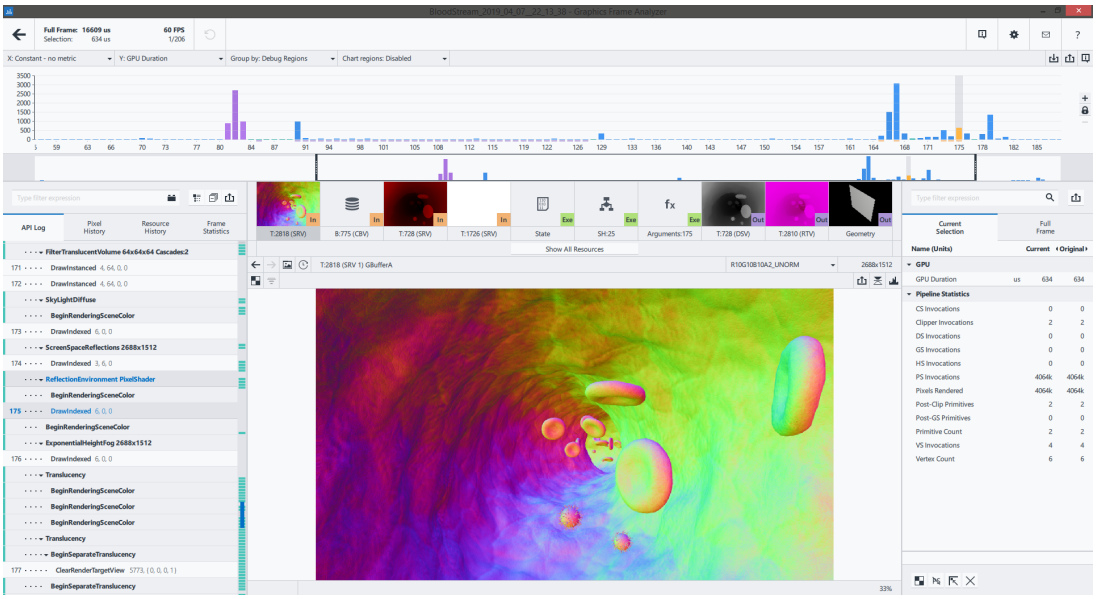
Pro ještě detailnější analýzu jednotlivých částí scény lze rovněž využít sadu nástrojů Intel Graphics Performance Analyzers. Jedná se externí sadu nástrojů, která není součástí Unreal Enginu, nicméně v rámci spolupráce Intelu a Epic Games má Unreal Engine oficiální podporu. Podobně tomu je i v případě enginu Unity, který je rovněž podporován.

Proces analýzy je prováděn pouze pro konkrétní jednotlivý snímek, nejedná se tedy o kontinuální záznam v čase. Je nutné však analyzovat běh programu na zkompilované verzi projektu (*packaged version*), protože se analyzátoru předá přímo cesta k .exe souboru, který má analyzovat.

Jednou z výhod tohoto nástroje je například zobrazení kompletního seznamu *Draw Calls*, který je možno třídit dle pořadí, dle doby zpracování (údaje jsou uvedeny v mikrosekundách), atd. Pro jednotlivé *Draw Calls* je také možné zobrazit obsahy jednotlivých bufferů, které byly v rámci daného *Draw Call* využity.

Další výhodou je například možnost zobrazení náročnosti pro jednotlivé typy světel, i v případě, že mají deaktivovány generování stínu. Tyto detaily nejsou přístupny dokonce ani v `GPU profiller` v samotném Unreal Enginu. V podstatě tak lze analyzovat, každý prvek scény od Post Proces efektů, přes osvětlení, až po geometrii.

Ukázka uživatelského rozhraní nástroje, kde byla analyzována scéna krevního řečiště je zachycena na obrázku 17.



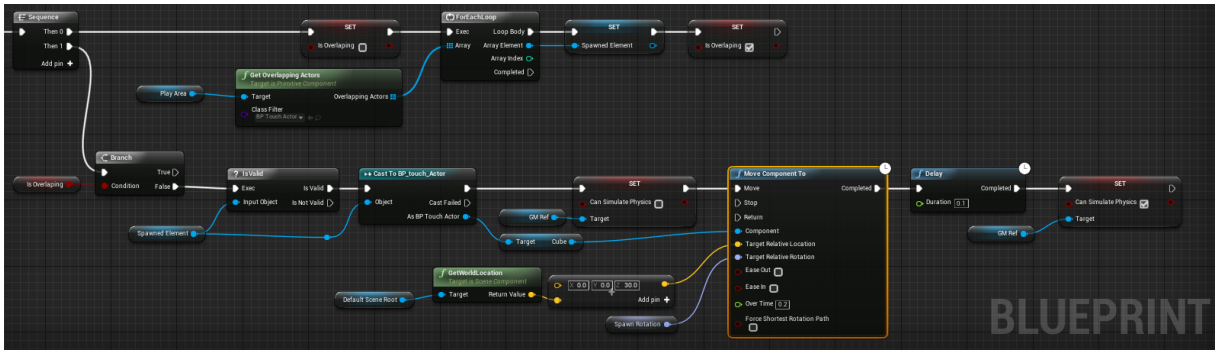
Obrázek 17: Ukázka uživatelského rozhraní nástroje Intel Graphics Performance Analyzers při analýze náročnosti scény krevního řečiště

# 5 Skriptování v Unreal Enginu

Unreal Engine 4 využívá tzv. vizuální skriptování. Logika dané aplikace je vytvářená mechanismem propojování jednotlivých uzlů proměnných, funkcí, operátorů atd. Interně se tyto skripty nazývají jako `Blueprints`. Základní principy při tvorbě skriptů jsou v zásadě stejné jako v případě psaného kódu v textovém editoru, pouze kód je zde vizualizován graficky.

Stejně jako u mnoha běžných skriptovacích jazyků, využívají blueprinty objektově-orientovaný přístupu tříd a objektů. Díky tomu umožňují vytvářet i pokročilé skripty bez zásahu do C++ kódu. Tvorba samotných blueprintů probíhá v UE editoru, takže není nutné využívat žádný externí nástroj. I přes to, že je tento systém již sám o sobě extrémně flexibilní, může být navíc rozšířen i C++ kódem. To v důsledku umožňuje vytváření základních systémů v C++, které mohou být následně rozšířeny v blueprintech [3].

Na obrázku 18 je vidět ukázka blueprintu, který má za úkol zjistit, zdali je hledaný prvek uvnitř hrací plochy. Pokud není, plynule ho přemístí zpět na požadovanou pozici.




Obrázek 18: Ukázka blueprintu pro plynulé přemístění objektu na určitou pozici

Blueprinty lze roztřídit dle typu rodičovských tříd, ze které dědí. Nejčastěji používanými typy jsou `Actor`, `Pawn`, `Character`, `PlayerController` nebo `Game Mode`.

Mimoto obsahuje ještě každý level tzv. `Level Blueprint`, který je na něj přímo navázán. Tudíž je veškerý jeho obsah a funkcionalita limitována pouze na daný konkrétní level.

Každý blueprint obsahuje tzv. `Event Graph`, ve kterém je obsažena logika celého prvku. Při jeho vytvoření se automaticky vygenerují tři základní události (*events*) - `Event BeginPlay`, `Event ActorBeginOverlap` a `Event Tick`. Z nichž lze následně „vytáhnout“ tzv. `Execution Wire`, který slouží k volání dalších částí skriptu.

U některých blueprintů se ještě navíc setkáváme s pojmem `Construction Script`. Tento skript se automatický vykoná při vytvoření nebo vložení blueprintu do scény. Slouží primárně k inicializaci, nebo vytváření komponent.

Komponenta je speciální typ objektu, který lze k blueprintu připojit. Takto lze připojit řadu různých prvků (mesh, kolizní objekt, částicový efekt, kameru, audio, světelný zdroj, atd.)[3].

# 6 Vývoj aplikací pro virtuální realitu

S nástupem virtuální reality má Unreal Engine 4 (od verze 4.10) rovněž podporu pro tento typ zařízení. Podporuje řadu platforem mezi které patří např. SteamVR, Oculus Rift, Playstation VR, Windows Mixed Reality, Samsung Gear VR, atd. Aplikace demonstrované v této práci jsou postaveny na platformě SteamVR, konkrétně pro zařízení HTC Vive.

Vývoj pro tento typ zařízení je do velké míry podobný vývoji aplikací pro PC s monitorem, myší a klávesnicí. Rozdíl spočívá ve využití `Motion Controller Pawn`, místo např. `First Person` nebo `Third Person character`. Tento `Controller` má již naimplementovanou základní funkčnost (přenášení pohybu headsetu a ovladačů do scény, jednoduchou teleportaci apod.).

Hlavním problémem vývoje aplikací pro virtuální realitu je, i přes již relativně vysoký výkon grafických karet a procesorů, náročnost samotného vykreslování. Brýle HTC Vive disponují dvojicí OLED displejů s celkovým rozlišením 2160×1200 px (tzn. 1080×1200 px pro oko). I když se může zdát, že je toto rozlišení dostačující, pro zobrazení drobných detailů stále nestačí. Z tohoto důvodu se do brýlí většinou posílá obraz ve větším rozlišení, které se následně zmenší (*downsampling*) na rozlišení daného zařízení. I přesto, že je výsledné zobrazované rozlišení stále stejné, výsledný obraz v brýlích je ostřejší a texty jsou čitelnější. Ve finále si tedy není těžké představit, že vykreslování obrazu pro každé oko, např. v dvojnásobném rozlišení v kombinaci s obnovovací frekvencí 90 Hz, je velice náročné na výpočetní výkon. Z těchto důvodů hraje optimalizace v aplikacích pro virtuální realitu velkou roli.

Pro tuto platformu byly vytvořeny aplikace s virtuální laboratoří, krevním řečištěm a anatomií člověka. Následující podkapitoly popisují tvorbu těchto aplikací a nabízí řešení problémů, které při implementaci vznikly.

## 6.1 Virtuální laboratoř

V laboratoři uživatel hravou formou získá základní znalosti z problematiky mikrobiologie. Základním požadavkem bylo vytvořit virtuální laboratoř, ve které se bude moci hráč volně pohybovat, učit se novým poznatkům a rovněž testovat své znalosti. Kompletní prostředí laboratoře, včetně všech přístrojů, vycházelo ze vzhledu současných moderních laboratoří a dodaných předloh. Ukázka výsledného prostředí laboratoře viz příloha B.

Scéna laboratoře byla vytvořena za pomocí modulárních objektů. Jedná se o efektivní způsob tvorby scény, kdy s relativně malým počtem dobře navržených 3D objektů lze vtvořit komplexně vypadající prostředí [26].

Ve scéně bylo rovněž využito `Lightmaps`. Díky tomu bylo nasvícení objektů uloženo do textur, což vedlo k ušetření výpočetního výkonu a také k přesnějšímu nasvícení celé scény (měkké stíny, difuzní odrazy světla, atd.). Jelikož se jedná o interiérovou scénu, kde skrze okna proniká venkovní světlo do interiéru, bylo zde využito také `Lightmass Portals`. To se ve výsledku projevilo přesnějším vnitřním nasvícením, bez jakýchkoliv viditelných artefaktů.

Celé prostředí je interaktivní. Uživatel se může s pomocí ovladačů pro virtuální realitu libovolně po laboratoři pohybovat, manipulovat s objekty atd.

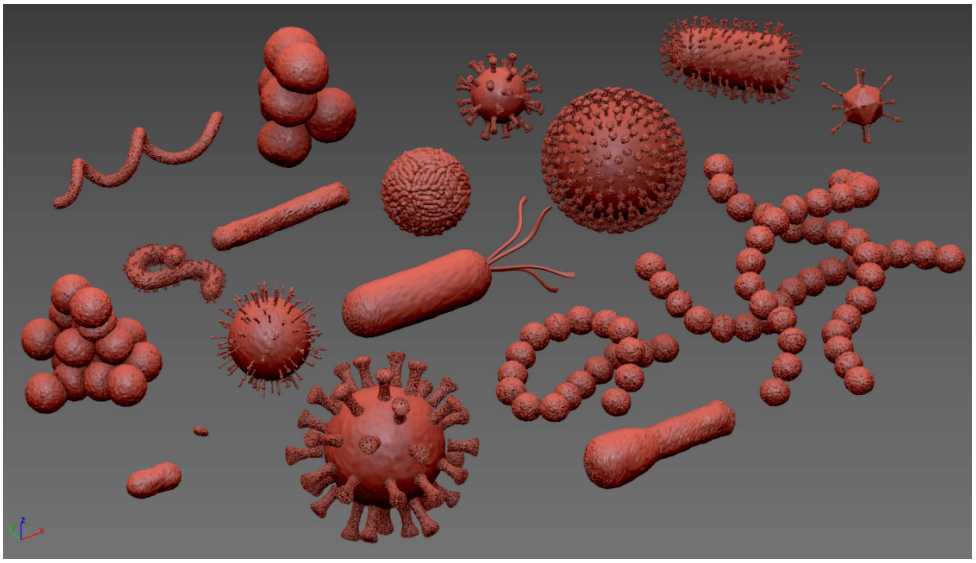
Prostor je rozdělen na dvě stanoviště. První stanoviště je věnováno samotné výuce virů a bakterií, druhá část slouží k prakticko-hernímu ověření znalostí dané problematiky. Část laboratoře byla ponechána volná, pro možná budoucí rozšíření.

### 6.1.1 Tvorba elementů

Po vytvoření prostředí laboratoře, bylo třeba se zaměřit na tvorbu jednotlivých elementů. Pro jejcih správnou interpretaci bylo nejprve nutno nastudovat danou problematiku [27], [28]. Následně byly vymodelovány jednotlivé 3D modely, tak aby vzhledem co nejvíce odpovídaly reálným obrázkům, (referencí byly fotografie z elektronových mikroskopů). Ukázka výsledných 3D Modelů viz obrázek 19.

Při vytváření těchto modelů bylo nutno vyřešit velikostní rozdíl mezi viry a bakteriemi (viry jsou řádově mnohem menší než bakterie). Aby mohl uživatel třídit elementy do těchto dvou kategorií, bylo nutno nalézt kompromis mezi realitou a hratelností v podobě úpravy velikostí. Změny ve velikostech bylo nutné provést i z důvodu nízkého rozlišení *OLED* panelů HTC Vive, protože některé elementy by byly příliš malé na to, aby je uživatel ve virtuální realitě viděl.

Poměry velikostí v rámci třídy elementů byly však zachovány (velikosti bakterií vůči sobě odpovídají, stejně tak je tomu i u virů). Skutečné a modifikované rozměry pro hru jednotlivých elementů reprezentuje tabulka 2.



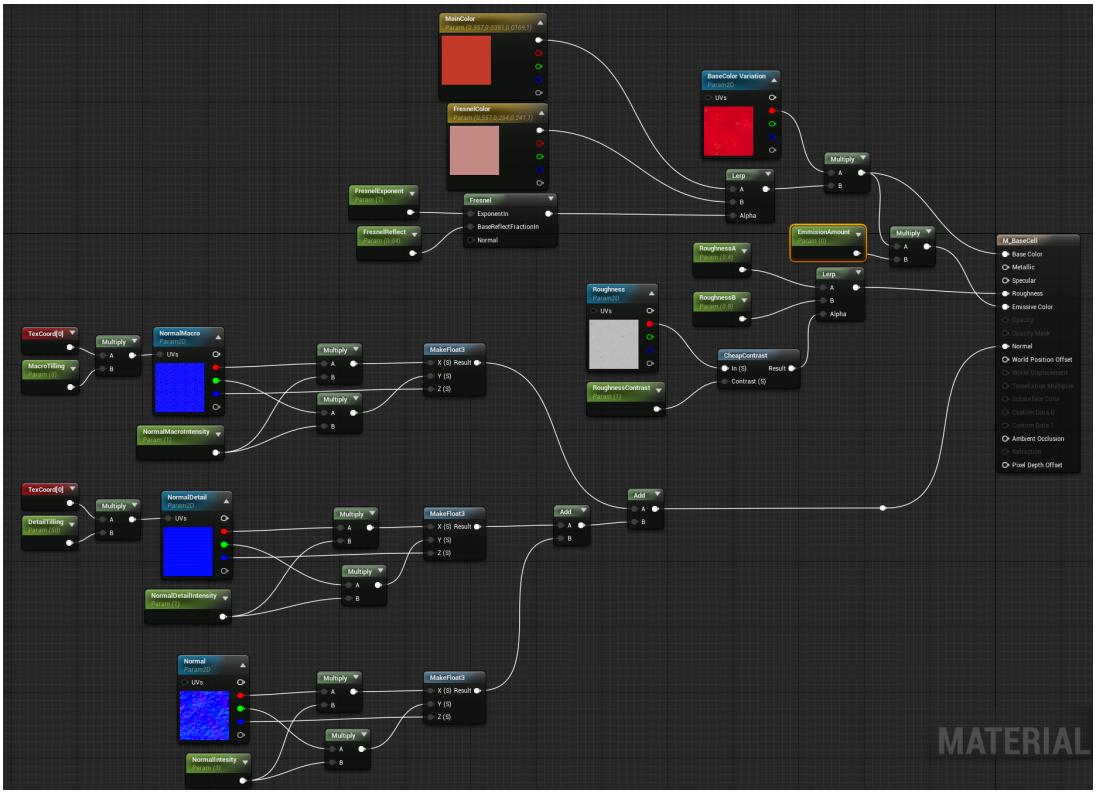
Obrázek 19: 3D modely virů a bakterií (ilustrativní zobrazení)

Tabulka 2: Tabulka velikostí virů, bakterií a krevních elementů

| Element | Reálná velikost ($\mu m$) | Velikost pro hru ($\mu m$) |
|---|---|---|
| Adenovirus | 0,09 - 0,10 | 1,90 |
| Ebola | 0,08 | 1,60 |
| Hepatitis A (Žloutenka typu A) | 0,06 - 0,07 | 1,30 |
| Rabies (Vzteklina) | 0,18 | 3,60 |
| SARS | 0,10 | 2,00 |
| Influenza A (Chřipkový virus) | 0,08 - 0,12 | 2,00 |
| Zika | 0,05 | 1,00 |
| Bordetella pertussis (černý kašel) | 0,20 - 0,80 | 0,96 |
| Borrelia burgdorferi (borelióza) | 5,00 - 20,00 | 6,00 |
| Clostridium botulinum (botulismus) | 3,00 - 4,00 | 4,20 |
| Escherichia coli | 2,00 - 3,00 | 3,00 |
| Staphylococcus aureus (zlatý stafylokok) | 1,00 - 1,50 | 1,50 |
| Streptococcus pyogenes (angína bakt. původu) | 0,50 - 2,00 | 2,40 |
| Mycobacterium tuberculosis (tuberkulóza) | 2,00 - 4,00 | 3,60 |
| Červená krvinka (Erytrocyt) | 7,00 | 7,00 |
| Bílá krvinka (Lymfocyt) | 7,00 | 7,00 |
| LDL Cholesterol | 0,02 | 0,35 |

Vytvořené 3D modely byly následně naimportovány do UE. Kolizní objekty byly

vygenerovány automaticky pomocí Unreal Enginu. Textury pro tyto modely byly vytvořeny ručně v nástroji Substance Designer. V Unreal Enginu byl následně vytvořen základní shader, ze kterého byly vytvářeny instance pro jednotlivé elementy. Pro větší kontrolu nad výsledným povrchem byl shader rozšířen o možnosti nastavení velikosti textury (*tilling*), výsledné barvy, intenzity *Normal* či *Roughness* mapy a intenzity Fresnelova členu (pro simulaci průsvitnosti na okrajích). Kromě toho byl rovněž přidán slot pro tzv. *Detail Normal* mapu. Ukázka grafu výsledného shaderu v Unreal Editoru - viz obrázek 20.

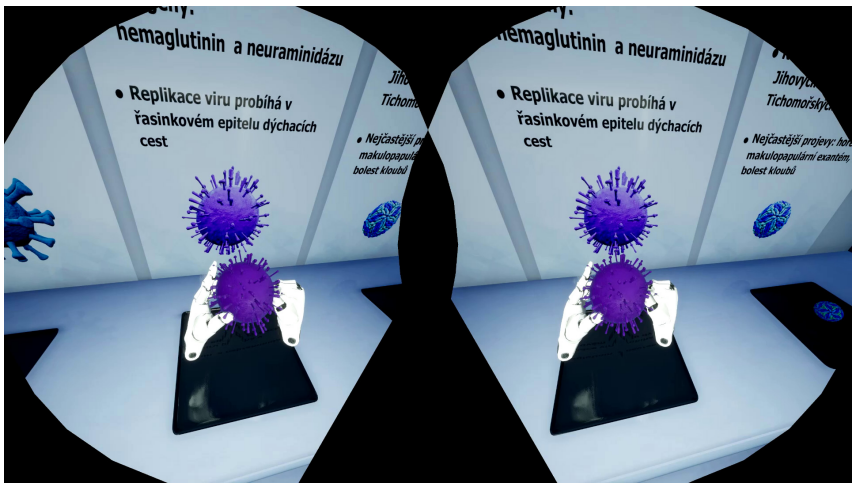



Obrázek 20: Graf základního materiálu pro cévu a krevní elementy

### 6.1.2 Výuková část

První část laboratoře je zaměřená na výuku v dané oblasti. Nacházejí se zde plakáty s popisem a zvětšené 3D modely virů a bakterií (viz obrázek 21). Pomocí ovladačů je lze rovněž uchopit do ruky a prohlédnou si je z blízka. Ukázka viz obrázek 22. Během testování se však ukázalo, že tato možnost s sebou přináší i komplikace. Občas se stalo, že uživatel element buď upustil, nebo odhodil. Z toho důvodu bylo nutné zajistit, aby se element po upuštění automaticky vrátil zpět na své původní místo. Řešením bylo využití funkce `Move Component To`, která v určeném čase interpoluje mezi současnou pozicí objektu a zadanou pozicí pro přemístění (výchozí pozice objektu uložená do lokální proměnné při spuštění hry). Výsledkem tedy byl plynulý návrat elementu na jeho místo v případě upuštění.


Obrázek 21: Výukové stanoviště laboratoře



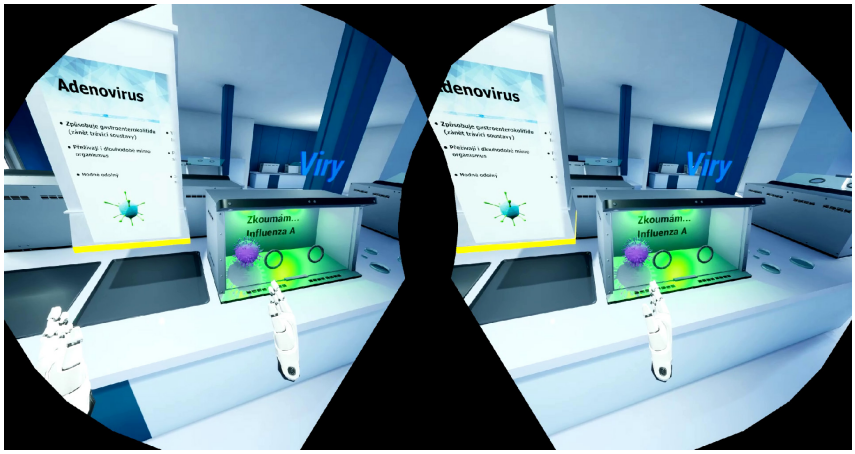

Obrázek 22: Uchopení objektu pomocí ovladačů

### 6.1.3 Herní část

Po bližším poznání jednotlivých virů a bakterií se uživatel přesune ke stanovišti, kde má možnost si své nabyté znalosti ověřit. Jeho úkolem je třídit přicházející elementy, na viry a bakterie podle tvaru, který si zapamatoval. Jak již bylo zmíněno, v praxi jsou viry podstatně menší než bakterie a pro roztřídění by stačila velikost. V našem případě bylo nutné zapamatovat si vzhled daného elementu podle konkrétního 3D modelu. Ukázka třídění viz obrázek 23.

Vytváření objektů ve scéně zajišťuje blueprint `BP_SpawnTube`, který z vloženého pole všech elementů náhodně vybere jeden a vloží ho do scény.

Uživatel dále pomocí ovladače element uchopí a přemístí ho na odpovídající místo - bakterie na podložku s Petriho miskami, viry do digestoře. Jak podložka, tak digestoř, jsou samostatnými blueprinty, které obsahují `Box Collision Component`, pomocí kterého kontrolují, jaké objekty se v daném prostoru nachází. Z pohledu implementace je vir či

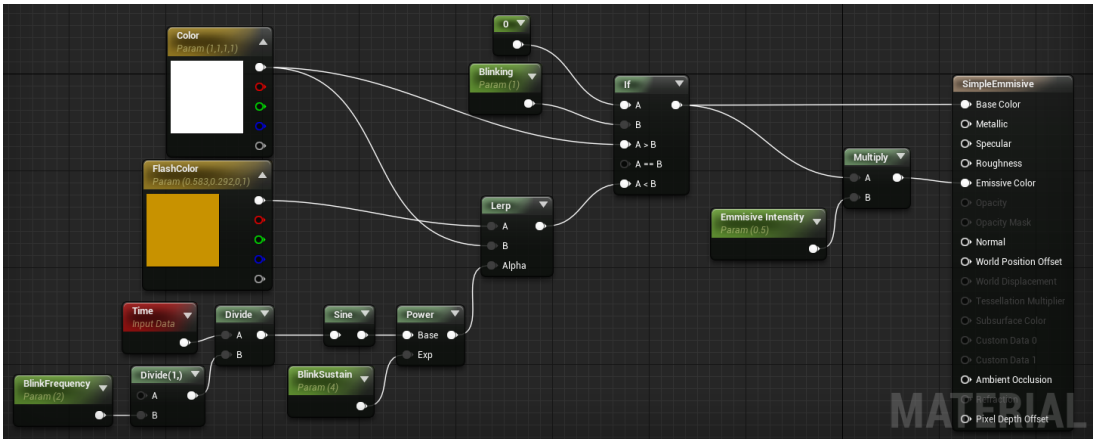



Obrázek 23: Ukázka třídění virů a bakterií

bakterie potomkem třídy `BP_Virus`, respektive `BP_Bacteria`. Rozpoznávání pak funguje na principu zjištění rodičovské třídy daného elementu.

Proces samotného rozpoznávání je signalizován barevně pomocí indikátoru na středovém sloupku. Jeho barvu je možno měnit díky využití parametrických vstupů materiálu. Díky nim lze za běhu měnit parametry, bez nutnosti rekompilace shaderu. Pro ovládání parametrů materiálu (konkrétně stavu blikání) ze samotného blueprintu je využita funkce `Set Scalar Parameter Value on Materials`, které je na vstup předán `Static Mesh`, název parametru materiálu a jeho hodnota.

V shaderu je pak blikání zajištěno pomocí parametru `Time` (vstup shaderu, který reprezentuje čas od spuštění programu) a funkce sinus. Zapnutí/vypnutí blikání je provedeno pomocí podmínky. Všechny důležité vstupy jsou parametrizovány, pro možnost ovládání. Graf zmíněného shaderu je zobrazen na obrázku 24.




Obrázek 24: Graf materiálu pro indikaci (blikání)

Po rozpoznání se uživateli zobrazí název daného elementu, a pokud element zařadil správně, element se rozloží (zmizí) a vygeneruje se nový.

## 6.2 Krevní řečiště

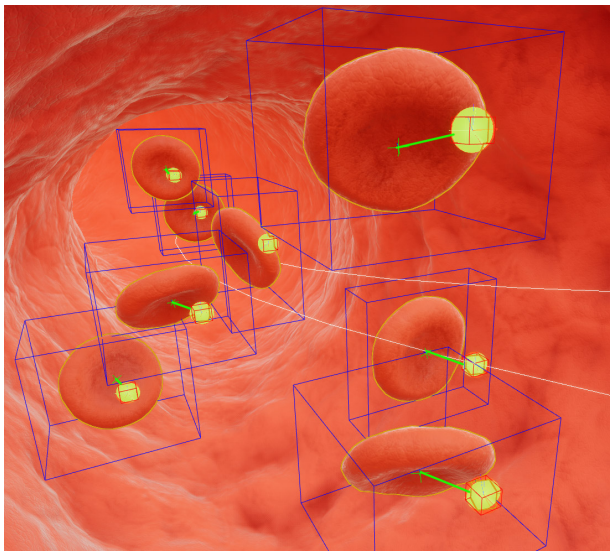
Hráč se pomocí virtuální reality přesune do samotné cévy, ve které proudí červené a bílé krvinky, ale zároveň i viry a bakterie. Cílem hráče je pomocí dvou ovladačů, které představují pistole, zneškodnit všechny příchozí viry a bakterie. Za každý zničený vir nebo bakterii, hráč obdrží body. Naopak za zničení krvinky, nebo krevní destičky jsou body odečítány. S rostoucím skóre se rovněž zvyšuje náročnost - v cévě se vyskytuje více virů a rovněž rychlost proudění se zvyšuje. Cílem hry je dosáhnout určitého počtu bodů.

Díky ponoření se do virtuální reality poskytuje tato aplikace uživateli reálnou představu o tom, jak vypadají jednotlivé krevní elementy, viry a bakterie, což mimo jiné slouží k jejich lepšímu zapamatování.

Ukázka výsledné scény krevního řečiště je znázorněna na obrázku v příloze C.

### 6.2.1 Implementace pohybu elementů

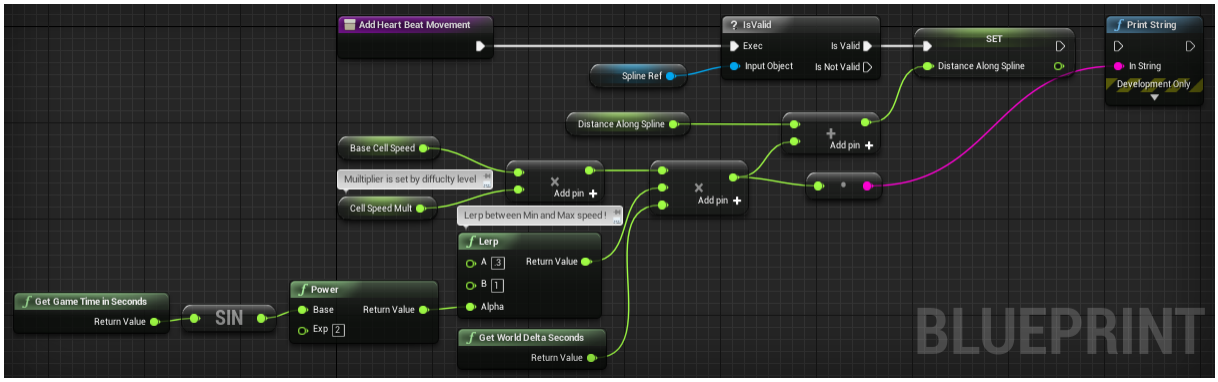
Základem všech elementů je blueprint `BP_ParentElement`, ze kterého poté dědí jednotlivé krevní elementy, viry a bakterie. Tento blueprint obsahuje tři důležité komponenty - vedoucí objekt (*Guide*), vedený objekt (*GuidedMesh*) a komponentu, která tyto dva objekty propojuje (`Physics Constraint`). Situaci ilustruje obrázek 25.



Obrázek 25: Znázornění vedoucího objektu (zelené body), jejich *Physics Contraints* (zelené čáry) a elementů (červené krvinky)

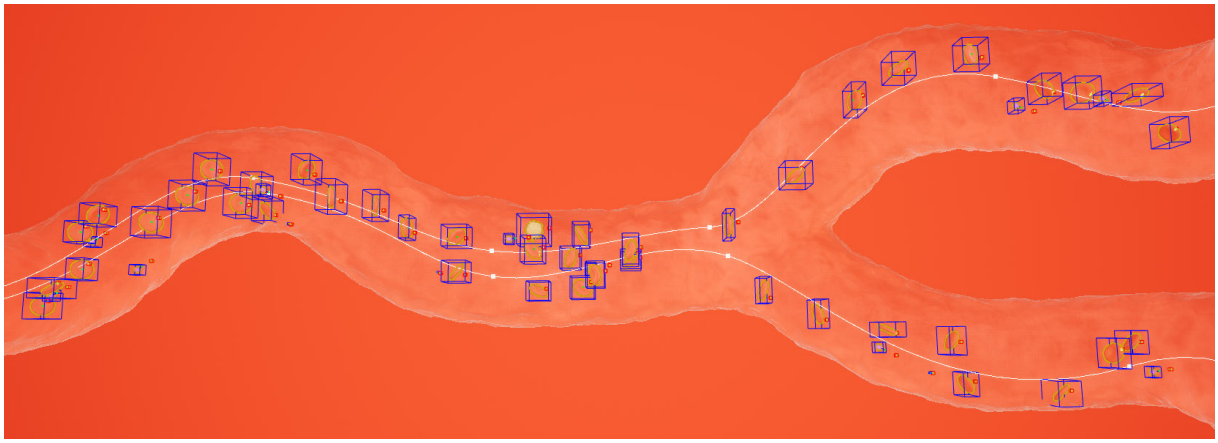
Samotný princip pohybu poté funguje částečně za pomocí křivek a fyzické simulace. „Vedoucí objekty“ jednotlivých elementů se pohybují po trajektorii zadané křivkou. Pro získání daného bodu na křivce bylo využito funkce `Get Location at Distance Along Spline`, která dle zadané vzdálenosti vrátí tříslożkový vektor s pozicí. Aby všechny body neležely přesně za sebou na křivce je k této pozici přidán náhodný posun v určitém rozmezí.

Aby pohyb lépe odpovídal realitě, dochází k neustálé změně jeho rychlosti, která reflektuje srdeční činnost (stah - systolu a uvolnění - diastolu). Tohoto efektu bylo dosaženo využitím funkce $f(x) = sin(x)^2$, kterou se rychlost pohybu násobí (viz obrázek 26). Tento pohyb je také doprovázen akusticky.




Obrázek 26: Využití funkce sinus pro dynamický pohyb elementů

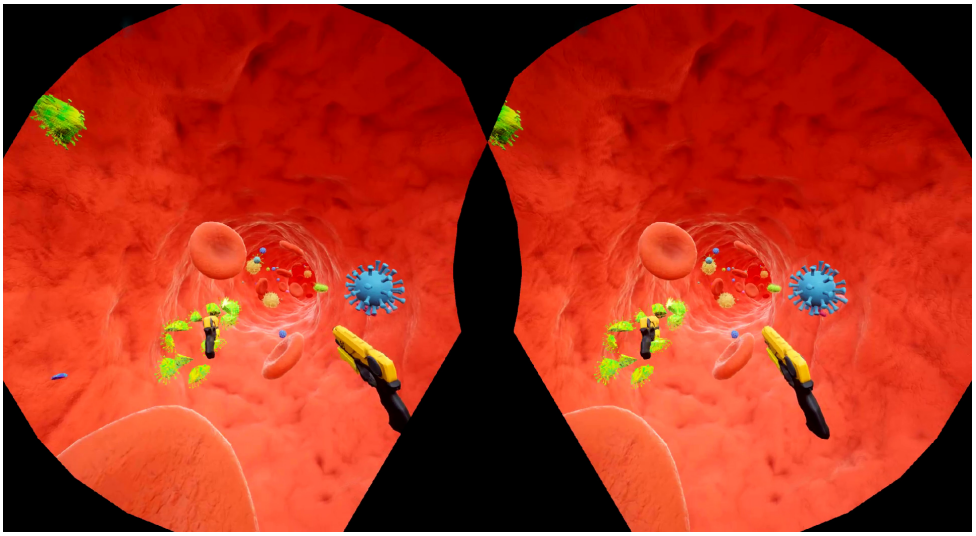
S využitím `Physics Constraint` [29] vedou poté tyto zmíněné „vedoucí objekty“ jednotlivé elementy cévou (viz obrázek 27). Jelikož mají tyto elementy zapnutou fyzickou simulaci, mohou adekvátně reagovat na kolize mezi sebou nebo se stěnou cévy. Takhle tyto elementy putují cévou, dokud nedojdou na konec, kde jsou tyto objekty následně odstraněny ze scény.



Obrázek 27: Céva a křivky, které znázorněné elementy následují

### 6.2.2 Implementace střílení

Jak již bylo zmíněno, úkolem hráče zde je zneškodnit viry a bakterie, které proplouvají cévou společně s červenými a bílými krvinkami. Ke střílení slouží uživateli ovladače, které v tomto případě reprezentují pistole. Stejně jako ostatní elementy v řečišti, byl tento model také vymodelován ručně. Textury byly následně vytvořeny v nástroji Substance Painter. Ukázka scény z pohledu uživatele viz obrázek 28



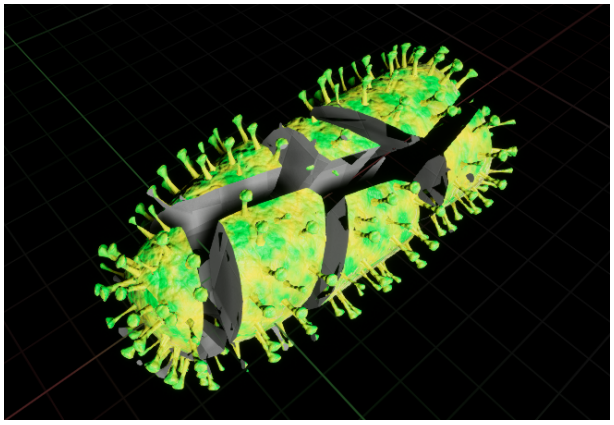
Obrázek 28: Ukázka scény krevního řečiště z pohledu uživatele

Po stisknutí spouště (*Trigger Button*) zavolá funkce `MyFireHandler` funkci `Spawn Actor BP_Projectile`, která do scény přidá `BP_Projectile`. Pohyb tohoto projektilu je zajištěn pomocí jeho komponenty `Projectile Movement Component`, což je komponenta, ke které se pomocí funkce `Set Updated Component` přiřadí druhá komponenta představující samotný mesh projektilu. Díky tomu se poté daný mesh projektilu může pohybovat. `Projectile Movement Component` umožňuje kromě rychlosti projektilu nastavovat také jeho gravitaci, odpor vzduchu, pravděpodobnost odrazu apod.

To, jestli projektil po výstřelu zasáhl některý z objektů ve scéně lze zjistit pomocí události `On Projectile Stop`, která vrací `Impact Result`. Díky tomu byla zjištěna přesná pozice zásahu, včetně normály a konkrétní zasažený objekt. Dále bylo testováno zdali je tento zasažený objekt potomkem třídy `BP_ParentElement`.

Každý element má proměnnou s jeho počtem „životů“. Viry a bakterie mají „život“ jeden, krvinky mají „životy“ dva. Je to z toho důvodu, že v případě nechtěného zásahu červené, či bílé krvinky, nebude uživatel hned penalizován ztrátou bodů. Kromě počtu životů má totiž každý element určen i počet bodů, které po jeho sestřelení uživatel obdrží. Body byly stanoveny dle náročnosti sestřelení jednotlivých elementů (určeny primárně jejich velikostí). Zatímco sestřelením viru či bakterie obdrží uživatel plusové body, sestřelením krvinek jsou uživateli body odečítány.

Při sestřelení elementů dojde k jeho rozpadu na malé části. Pro tento efekt je využito pluginu `Nvidia APEX PhysX` [30]. Tento plugin umožňuje před-fragmentovat daný objekt přímo v samotném editoru enginu. Současná verze pluginu umožňuje fragmentaci pouze jedné úrovně- tzn. fragmentace fragmentovaných částí není podporovaná. Pro účely této aplikace byla však jednoúrovňová fragmentace dostačující. Počet na kolik částí má plugin objekt rozdělit jde samozřejmě také nastavit. Ukázka fragmentace daného viru je znázorněna na obrázku 29



Obrázek 29: Ukázka fragmentace objektu (virus vztekliny) pomocí pluginu Nvidia Apex

### 6.2.3 Načítání souboru s parametry

Aplikace je rozšířena o možnost nastavování parametrů chování ze souboru. Konkrétně se jedná o parametry, které definují jednotlivé úrovně ve hře. Mezi základní parametry, které lze pro jednotlivé úrovně nastavovat, patří rychlost proudění a rychlost vytváření elementů. Rovněž lze také nastavit bodový rozsah pro jednotlivé úrovně. Dále je možné nastavit pravděpodobnosti vytváření jednotlivých elementů pro konkrétní úroveň. Ukázka části konfiguračního souboru je znázorněna v příloze A.

Díky využití tohoto konfiguračního souboru lze detailně měnit chování celé aplikace, bez nutnosti její celé rekompilace. Tohoto přístupu bylo využito především s ohledem na způsob využití celé aplikace. Jelikož je aplikace určena primárně pro vzdělávací subjekty, umožňuje toto nastavení přizpůsobit si chování aplikace dle konkrétních potřeb či dané probírané látky. Rovněž je také možné vytvořit si sadu několika těchto konfiguračních souborů, které mohou odpovídat různým obtížnostem a díky tomu poté snadno přepínat mezi jednotlivými úrovněmi obtížností.

Jelikož Unreal engine načítání dat z vlastních souborů neumožňuje bylo využito pluginu `Rama's Victory Blueprint Library`. Tento plugin obsahuje funkci `Load String from File`, která umožňuje přistoupit ke konkrétnímu textovému souboru a načíst jeho obsah do proměnné datového typu string. Problémem samozřejmě je, že po načtení je nutné tento řetězec zpracovat. Z toho důvodu byl v Unreal enginu vytvořen vlastní *parser*, který ze souboru získá jednotlivé parametry a uloží patřičné hodnoty do lokálních proměnných.

## 6.3 Anatomie člověka

Poslední vytvořenou aplikací, pro virtuální realitu byla aplikace anatomie, která se konkrétně zaměřuje na znalost opěrné (kosterní) soustavy člověka. Z pohledu hratelnosti nabízí i tato aplikace dva módy - výukový a testovací.

Podkladem pro tuto aplikaci byl kompletní 3D model lidské anatomie, který byl zpracován a jednotlivé kosti byly naimportovány do UE. Samotné prostředí bylo v rámci zachování konzistence vytvořeno v podobném stylu jako virtuální laboratoř.

Ve scéně se kromě samotného modelu kostry nachází i virtuální tablet, který slouží k zobrazování jednotlivých informací. Ve výukovém módu, kde může uživatel volně uchopit jakoukoliv kost, se na tabletu zobrazují české a latinské názvy, společně s popisem, který uživateli přiblíží základní informace o konkrétní kosti. V testovacím módu slouží tablet k zobrazování názvů kostí, které má uživatel poznat (uchopit do ruky). Pro snazší ovládání je možné do ruky uchopit i tablet a přemístit si ho tak v prostoru dle potřeby. Text, který se na tabletu zobrazuje, je řešen využitím komponenty `Widget` v samotném blueprintu tabletu.

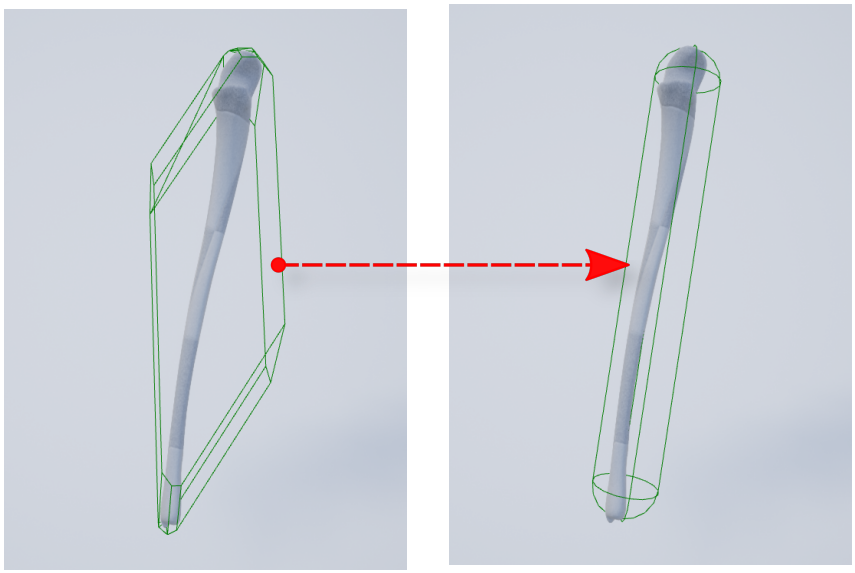
Co se týče materiálů, byla využitím `Dynamic Material Instance` kostem přiřazena náhodná barva z předem vytvořené barevné palety. Důvodem bylo, že bílé kosti nebyly oproti světlé místnosti dostatečně kontrastní a navíc pro některé uživatele nemuselo být jasné, kde daná kost začíná a kde končí. Barevným odlišením je každá kost zřetelně odlišena.

Jedním z problémů, ke kterému docházelo, byla problematická klasifikace objektu (kosti) který se má při přiblížení ovladače (virtuální ruky) uchopit v případě, kdy jsou dva objekty blízko sebe. Problém byl převážně v neoptimálně vytvořených kolizních objektech jednotlivých kostí. Tyto kolizní objekty byly vygenerovány automaticky, společně s importem objektů. Ukázalo se však, že v některých případech neobepínají samotný model (mesh) dostatečně těsně, a kolizní objekty se u některých kostí překrývají. Díky tomu se někdy stávalo, že došlo k uchopení kosti, kterou uživatel neměl v úmyslu uchopit.

Řešením tohoto problému bylo ruční vytvoření kolizních objektů u problematických případů. Ukázka nevhodně vygenerovaného kolizního objektu a jeho následná opravená verze je znázorněna na obrázku 30. Ukázalo se tedy, že i když Unreal Enginu tuto funkcionalitu nabízí, nemusí vždy poskytovat ideální řešení.

Pro ještě větší zpřesnění úchopu byl dále upraven i kolizní objekt samotného ovladače (virtuální ruky). Původní kolizní objekt byl totiž v poměru k ruce velký, což mohlo usnadnit uchopení objektu v případech, kdy je objekt od ruky vzdálen. Tato vlastnost však byla v tomto případě kontraproduktivní, poněvadž kosti jsou relativně blízko sebe a neumožňovala tak přesný a jistý úchop konkrétní kosti. Problém byl tedy vyřešen zmenšením tohoto kolizního objektu a zároveň jeho lepším umístěním do prostoru dlaně.

Pro zobrazování informací se ve scéně, kromě tabletu, nachází i velkoplošná tabule. Tato tabule není primárně určena pro samotného uživatele, ale spíše pro pozorovatelé, kteří dění ve virtuální realitě sledují zprostředkovaně přes monitor. Důvodem tohoto řešení byla především



Obrázek 30: Anatomie - úprava kolizních objektů pro vyřešení problému s úchopem

velikost textu na tabletu, který je sice dobře čitelný pro samotného uživatele, nicméně pro publikum pozorující monitor z větší vzdálenosti ve spojitosti s neustálou manipulací tabletu je již text čitelný hůře. Dalším problémem je rovněž ořez obrazu pro monitor v případě, že nechceme zobrazovat jednotlivé obrazy pro každé oko. Unreal Engine nabízí primárně dvě možnosti ořezů, a to horizontální a vertikální, kde je ořez proveden vždy z levého obrazu. Je však patrné, že díky tvaru vykreslovaného obrazu, musí být ořez poměrně razantní a z tohoto důvodu se může stát, že i když uživatel ve virtuální realitě tablet vidí, publikum sledující duplikovaný obraz na monitoru, tablet již vidět nemusí. Tímto ořezem je pochopitelně ovlivněn i celý zorný úhel, což se rovněž negativně projevuje omezenou přehledností pro sledující publikum.

Z těchto důvodu bylo tedy přikročeno k řešení, kdy je obraz pro diváky vykreslován zvlášť z jiného, statického úhlu, který zabírá jak kostru, tak tabuli. Na základě testování se nakonec potvrdilo, že se v tomto případě jedná o lepší řešení, než duplikovat zobrazení virtuální reality na monitor.

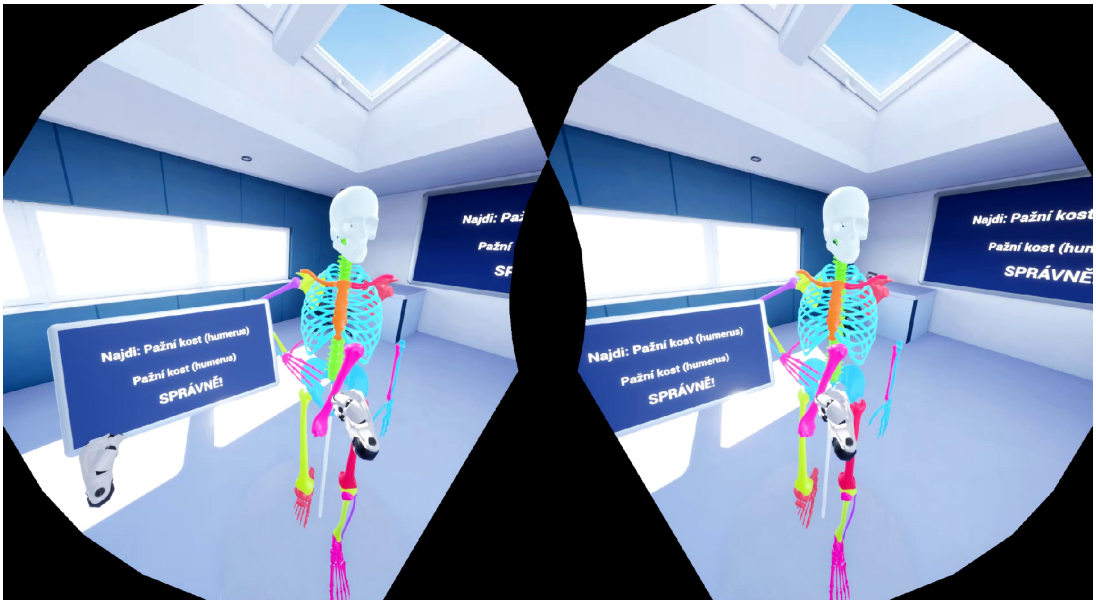
V případě, že by bylo nutno z nějakého důvodu obraz duplikovat, je možné tuto možnost vypnout pomocí argumentů příkazové řádky při spuštění aplikace. Zadáním argumentu `noCam` nebude na monitor vykreslován obraz ze statického záběru, ale bude se klasicky duplikovat z HMD.

Ukázka scény z pohledu kamery diváka, respektive pohledu uživatele je znázorněna na obrázku 31 a 32.

Princip vykreslování odlišného obrazu na monitoru, než v headsetu, je dosažen využitím tzv. `Render Target`, který snímá obraz podobně jako kamera s tím, že je ale tento obraz ukládán do textury, která se pak následně vykresluje na obrazovku monitoru.

Nevýhodou tohoto přístupu je samozřejmě nutnost vykreslování scény z dalšího pohledu,


Obrázek 31: Anatomie - ukázka z pohledu diváka




Obrázek 32: Anatomie - ukázka z pohledu uživatele

což se může obecně negativně projevit na vykreslovací frekvenci. V tomto případě je však scéna relativně nenáročná, už jen díky malému počtu objektů, což má přímý dopad na počet *Draw Calls*, který je díky tomu relativně nízký (průměrně okolo 300). Z pohledu nasvícení je ve scéně využito rovnoběžného `Directional` světelného zdroje, které je typu `stationary`, takže využívá `UE Lightmass` pro uložení nasvícení scény do `Lightmap`.

# 7 Vývoj aplikací pro smart tabule a dotykové monitory

Kromě platformy virtuální reality byla věnována pozornost i zařízením s dotykovou vrstvou. Mezi taková zařízení patří např. tablety, dotykové monitory, či smart tabule. Právě s pojmem smart tabule se stále častěji setkáváme i ve školách, kde zefektivňují a obohacují výuku.

Tvorba aplikací pro tato zařízení se od klasických desktopových aplikací (pro myš a klávesnici) liší v několika ohledech. Klasická myš a klávesnice je pochopitelně nahrazena pouze dotyky prstů či pera. K tomu však musí být uzpůsobeno i uživatelské rozhraní, od kterého se očekává podobné chování, jako známe například z chytrých mobilních telefonů. Rovněž musí být brán ohled na velikost zobrazované plochy. Např. v případě velkoplošné smart tabule je důležité správné rozmístění a velikost ovládacích prvků tak, aby zajišťovaly uživateli co nejlepší ergonomii ovládání.

Pro tento typ zařízení byla vytvořena aplikace, která si klade za cíl seznámit uživatele se základními informacemi o konkrétních virech a bakteriích. Aplikace se dělí na dvě části a to na výukovou část a prakticko-herní část, kde si uživatel ověří kompetitivní formou nabyté znalosti.

Cílem nebylo vytvořit realistickou simulaci ale spíše přehlednou a stylizovanou formou podat uživateli základní informace o problematice. Důraz byl kladen na jednoduchost ovládání a celkovou plynulost pohybů a animací, tak aby výstupná aplikace působila příjemným a uhlazeným dojmem. Program je opět navržen modulárně s ohledem na snadné případné úpravy či rozšíření.

## 7.1 Tvorba uživatelského rozhraní

Důležitým bodem každé aplikace je navržení uživatelského rozhraní, které je intuitivní a umožní uživateli snadnou a příjemnou navigaci v aplikaci. Pro tvorbu uživatelského rozhraní bylo využito tzv. `Unreal Motion Graphics UI Designer` (`UMG`). Díky navázání na blueprinty umožňuje tento nástroj tvorbu komplexních uživatelských rozhraní, která efektivně propojují ovládací prvky s objekty ve scéně.

Celé uživatelské rozhraní je tvořeno s ohledem na proměnlivou velikost rozlišení smart tabulí či monitorů. Je tedy tzv. responzivní, schopno přizpůsobit se velikosti (a rozlišení) obrazovky. Díky tomu lze výslednou aplikaci pohodlně používat nejen na tabulích ale i na klasických monitorech s dotykovou vrstvou.

Ohled byl brán také na barevná schémata. V aplikaci figurují primárně dvě barvy - modrá a oranžová. Tyto barvy jsou vůči sobě komplementární a vhodně se tak doplňují. Napříč celou aplikací jsou tyto barvy následně asociovány s jednotlivými typy elementů. Bakterie jsou vždy vyobrazovány na modrém pozadí, naopak viry na oranžovém. Bylo tak učiněno z toho důvodu, aby si uživatel mohl vizuálně spojit daný element s konkrétní barvou, což by mělo ve výsledku usnadnit zapamatování a zefektivnit tak celý proces učení.

## 7.2 Výuková část aplikace a její implementace

Výuková část má podobu seznamu rozděleného na dvě poloviny - viry a bakterie. Kompletní seznam všech elementů je znázorněn na obrázku 33 (pochopitelně se jedná o ilustrativní obrázek. Ve výsledné aplikaci uživatel vždy vidí pouze jeden záznam - element).


Obrázek 33: Kompletní seznam všech elementů rozdělený na dvě poloviny - bakterie a viry

V tomto seznamu lze plynule posouvat pomocí klasického přejetí prstem, tak jak je obvyklé např. v současných uživatelských rozhraních v mobilních zařízeních. Jelikož Unreal engine nemá tuto funkcionalitu vnitřně implementovanou, bylo nutné ji ručně vytvořit. Listování je založeno na principu posouvání kamerou, nikoliv samotného seznamu. Vzhledem k situaci jde o efektivnější řešení, než hýbat se všemi elementy ve scéně.

Pro ovládání posuvu bylo nejprve třeba zjistit konkrétní pozici (pixel), na který uživatel v čase $t_0$ umístil prst. Pokud se v čase $t_1$ prst stále dotýká obrazovky a pozice se ve vodorovné ose $X$ změnila oproti pozici v čase $t_0$, spočte se absolutní hodnota rozdílu těchto hodnot. Pokud je tato hodnota větší než stanovený práh, dojde k posunu kamery v patřičném směru. Tento práh slouží k odlišení posunu prstu od obyčejného kliknutí. V případě, že by tam tento práh nebyl, uživateli by se mohla hýbat kamera i při klikání (což by bylo pochopitelně nežádoucí). Dále bylo nutné zajistit, aby kamera nezůstávala v „mezi-polohách“, tedy mezi jednotlivými listy seznamu. Po dokončení posuvu se proto zjistí pozice kamery a pomocí funkce `Move Component To` se plynule posune do polohy, která odpovídá nejbližšímu listu (elementu).

Každý „list“ v seznamu obsahuje kromě informací o daném viru či bakterii také její interaktivní 3D model, s kterým lze libovolně natáčet a je tak možné si ho prohlédnout ze všech stran. Tento prvek se snaží o to, aby si uživatel vytvořil prostorovou představu o daném elementu, což by rovněž mělo usnadnit jeho zapamatování. Směr a rychlost otáčení je stanoven tím, jak rychle a v jakém směru uživatel prstem „přejede“ přes element. Otáčení funguje na podobném principu jako posun v seznamu, zde však je posun převeden na rotační pohyb využitím funkce `Make Rotator`. Toto řešení však nezajišťuje setrvačnost pohybu. Ta je dodatečně řešena pomocí funkce `Add Angular Impulse in Radians`.

Během procesu testování bylo zjištěno, že by bylo vhodné mít v seznamu také možnost „přeskakovat“ (původní verze umožňovala pouze posun o jeden prvek vpřed, nebo vzad). Z toho důvodu byla do uživatelského rozhraní přidána navigační lišta (viz obrázek 34), která tento problém eliminuje a rovněž vizualizuje, kde se v seznamu uživatel momentálně nachází.

Obrázek 34: Navigační lišta pro rychlé listování v seznamu elementů

Ukázka tří listů (obrazovek) z výukové části aplikace viz obrázek 35.


Obrázek 35: Ukázka třech obrazovek (listů) výukové části aplikace

## 7.3 Herní část aplikace a její implementace

Tato část slouží k praktickému ověření znalostí nabytých ve výukové části. Uživatel je opět přemístěn do virtuální laboratoře, kde je jeho úkolem třídit příchozí elementy na viry a bakterie. Prostředí vychází ze stejné scény laboratoře, jako tomu bylo u verze pro virtuální realitu. Bylo však upraveno rozmístění určitých prvků, tak aby lépe vyhovovalo stylu ovládání. Rovněž bylo nutné vytvořit novou logiku hry, která je uzpůsobena dotykovému ovládání. Ukázka herního prostředí viz obrázek 36.


Obrázek 36: Ukázka prostředí herní části aplikace

Úkolem je tedy roztřídit postupně objevující se viry a bakterie. Manipulace s elementy probíhá formou *Drag and Drop*. Uživatel prstem přetáhne element na odpovídající podložku pro daný typ - podložka pro viry, nebo podložka pro bakterie. Pokud přemístí element na adekvátní místo, zobrazí se název daného elementu, element zmizí a objeví se další.

Z pohledu implementace je základním prvkem scény `BP_Touch_Actor`. Z něho pak dědí třídy `BP_Virus` a `BP_Bacteria`. Samotné elementy (které jsou rovněž jednotlivé blueprinty) pak dědí z těchto dvou tříd. Vytváření samotných objektů ve scéně zajišťuje `BP_Spawn_Manager`, který z předem vytvořeného pole elementů vybere náhodný prvek a přidá ho do scény. Vyloučením

daného indexu je rovněž zajištěno, že nedojde k opakovanému vložení stejného elementu do scény.

Po vytvoření a vložení elementu do scény bylo třeba zajistit možnost jeho manipulace. Bylo tedy třeba naimplementovat již zmíněný *Drag and Drop* systém.

Nejprve bylo nutno zjistit pozici ve *Screen Space* - na jaký konkrétní pixel uživatel umístil prst. Dále je z kamery proveden *Line Trace*, pro zjištění pozice, kde se tento paprsek protíná ve scéně (*World Space*). Pokud je zasažený objekt potomkem z třídy `BP_touch_Actor`, bylo ještě následně vyhodnoceno, zdali se jedná o vir, nebo o bakterii. K vizuální indikaci uchopení se element pomocí jednoduché animace „přiblíží“. Pokud uživatel element neupustí, jsou prováděny další *Line Trace*, které slouží k zjištění pozice, na kterou se má element posouvat.

Aby se zabránilo nechtěnému přesunutí objektů mimo stůl, byla herní plocha, po které lze objekty přesouvat, omezena. Nicméně i přes limitovanou herní oblast pro posuv se během testování občas stalo, že uživatel prvek „vyhodil“ a vlivem setrvačnosti prvek opustil oblast. Tento problém byl vyřešen přidáním kolizního objektu a následným testováním, zdali se element nachází uvnitř tohoto objektu. Pokud ne, objekt se plynule přemístí na výchozí pozici. K plynulému přemístění bylo využito funkce `Move Component To`, která v zadaném čase interpoluje pozici objektu se zadanou pozicí, pro přemístění.

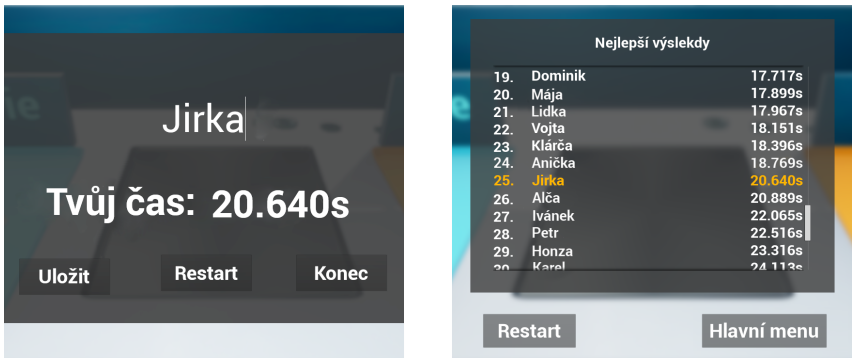
Klasifikaci přesunutého objektu na podložku zajišťuje `BP_Tray`. Tento blueprint obsahuje `Box Component`, pomocí kterého zjistí, jestli je přesunutý objekt potomkem třídy `BP_Virus`, nebo `BP_Bacteria`. Následně je provedena patřičná akce - pokud dědí element ze třídy, kterou podložka neakceptuje, podložka se zbarví červeně a k času se přičtou trestné vteřiny. V opačném případě se spustí jednoduchý částicový efekt, který indikuje, že se s element rozkládá a zároveň maskuje zmizení samotného elementu. `BP_Tray` poté informuje `BP_Spawn_Manager` a ten do scény vygeneruje další element. Takto se proces opakuje, do stanoveného limitu (10 prvků na hru). Po dosažení tohoto limitu se hra ukončí a uživateli je představena závěrečná obrazovka s vyhodnocením testu.

Na této obrazovce má uživatel možnost uložení svého výsledku. Zadáním jména, či přezdívky a stisknutím tlačítka uložit, dojde k vytvoření záznamu. Ukládání, respektive načítání, je v Unreal Enginu řešeno použitím funkcí `Save Game to Slot` a `Load Game from Slot`. Pro uložení bylo nutné vytvořit nejprve strukturu, která obsahuje dvě hodnoty - `String name` a `float time`.

V případě, že chce uživatel skóre uložit, musí se nejprve ověřit, zdali už soubor s uloženými pozicemi existuje. V případě, že neexistuje, je do dočasného pole vložen pouze jeden záznam (zadané jméno a skóre). Toto pole je následně předáno funkci `Create Save Game Object`, která vytvoří tzv. `Save Game Object`. Posledním krokem je předání tohoto objektu funkci `Save Game to Slot`, která data, pod zadaným jménem uloží na disk. Ukládání skóre v případě, že soubor s uloženými pozicemi již existuje, je do jisté míry podobný. Do dočasného pole je však nejprve nutné postupně načítat obsah celého souboru a podle uložených časů pak na správné místo vložit současný hráčův výsledek s tím, že časy jsou řazeny vzestupně.

Nakonec je toto dočasné pole opět předáno funkci `Create Save Game Object` a `Save Game Object` je dále předán funkci `Save Game to Slot` k uložení.

Po uložení se uživateli zobrazí tabulka výsledků, kde může vidět své umístění v porovnání s ostatním soutěžícím. Ukázka rozhraní pro ukládání skóre a následná tabulka nejlepších výsledků viz obrázek 37.

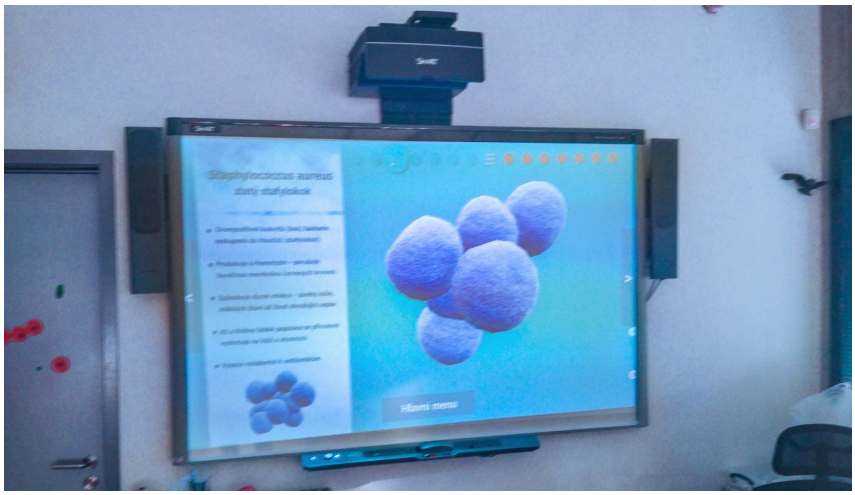



Obrázek 37: Rozhraní pro ukládání skóre a následná tabulka nejlepších výsledků

Jisté komplikace přineslo zadávání textu (jména) v případě nepřítomnosti fyzické klávesnice. Samotné kliknutí na textové pole v Unreal Enginu totiž nevyvolá klasickou virtuální klávesnici. Řešením tedy bylo manuální spuštění procesu `OSK.exe` (*On Screen Keyboard*). Tato funkcionalita (spuštění vlastních procesů) však není v samotném enginu možná. Bylo tedy opět využito pluginu `Rama's Victory Blueprint Library`, který umožňuje vytvářet jednotlivé procesy přímo skrze blueprinty.

Pro případ, že by však uživatel nechtěl virtuální klávesnici používat, byla doimplementována možnost deaktivace zmíněného procesu pomocí argumentů příkazového řádku. V případě použití argumentu `noOSK`, se tedy proces `OSK.exe` nespustí.

Ukázka výukové části aplikace ve Světě techniky Ostrava viz obrázek 38.



Obrázek 38: Ukázka aplikace ze Světa techniky

# 8 Závěr

Hlavním cílem této práce bylo ověřit možnosti využití moderních herních enginů v oblasti vzdělávání. Pro ověření byly vybrány oblasti, kde je výuka vedena tradiční formou, bez možnosti větší interakce mezi žákem a probíranou látkou. Použitím smart tabulí a virtuální reality dostává výuka úplně jiný rozměr, který usnadňuje celý proces učení a navíc je pro žáky atraktivnější. Výstupní aplikace, byly realizovány právě na těchto dvou platformách.

První skupina aplikací byla vytvořena pro zařízení virtuální reality. V prostředí virtuální laboratoře se uživatelé nejprve seznámí s problematikou mikrobiologie. Intenzivnímu vnímání problematiky napomáhá možnost uchopit si vytvořené modely bakterií a virů, prohlédnout si je zblízka a seznámit se s jejich vlastnostmi. Prostředí je navrženo tak, aby plnilo účel jednak z pohledu výuky, tak i z pohledu ověření nabytých znalostí formou správného zařazování daného modelu do kategorie virů či bakterií.

Další vytvořenou aplikací bylo krevní řečiště, které hravou formou přibližuje reálnou situaci uvnitř cév v lidském těle. Uživatel je vtažen do virtuální cévy, kde se setkává s prouděním krvinek, bakterií a virů napříč krevním řečištěm. Jeho úkolem je rozlišit jednotlivé elementy a odstranit viry a bakterie z řečiště. Z pohledu použití nabízí aplikace možnosti nastavení parametrů pro jednotlivé úrovně z externího souboru, což např. umožní přizpůsobit si aplikaci, dle konkrétní probírané látky, bez nutnosti rekompilace celé aplikace.

Poslední realizovanou aplikací, vytvořenou pro virtuální realitu, je scéna s modelem kostry lidského těla, která prověřuje znalosti uživatele z oblasti anatomie.

Z vytvořených aplikací je patrné, že virtuální realita má univerzální využití a lze ji tak aplikovat nejen na oblast biologie, ale prakticky na libovolnou oblast vzdělávání.

Druhou platformu zastupuje aplikace vytvořená pro použití na smart tabulích, které se v současné době začínají používat při různých formách výuky. Při vytváření dané aplikace byl velmi důležitý vhodný návrh uživatelského rozhraní, které je intuitivní a zároveň ergonomické.

Výukové téma virtuální laboratoře pro platformu virtuální reality i pro interaktivní smart tabule bylo obsahově podobné. Přesto bylo možné v praxi pozorovat různou intenzitu vnímání informací u jednotlivých uživatelů, v závislosti na zvolené platformě.

Aplikace byly testovány v laboratoři, prakticky ověřeny na dnech otevřených dveří školy apod. Na závěr se také testovala možnost nasazení ve výukových programech Světa techniky Ostrava. Poznatky získané během těchto testování dopomohly k odladění problémů, které nemusí být na první pohled, bez reálného nasazení, snadno odhalitelné.

Jedním z požadavků byla rovněž možnost dalšího rozšíření vytvořených aplikací. Díky konkrétnímu využití aplikací, primárně ve Světě techniky Ostrava, byly aplikace vytvořeny jako samostatně spustitelné programy (nikoliv jako jeden program slučující všechny části do jednoho). V konkrétním nasazení toto řešení tedy usnadní možnosti aktualizací, poněvadž

v případě rozšíření jedné části není nutné posílat celou (velkou) aplikací, ale pouze jednu danou, aktualizovanou část.

Unreal engine se ukázal jako ideální nástroj pro vytváření multiplatformních aplikací. Do budoucna lze doufat a očekávat, že aplikací, podobné těmto, bude ve výuce přibývat. Praktické ohlasy potvrzují, že využití virtuální reality má velký potenciál a může velmi zefektivnit výuku v libovolné oblasti.

# Literatura


[1] Dolní oblast VÍTKOVICE, z.s. *Svět techniky Ostrava - Science and Technology Center Ostrava* [online]. 2019, [cit. 2019-04-10]. Dostupné z: http://stcostrava.cz

[2] HTC Corporation, *VIVE$^{TM}$ / Buy VIVE Hardware* [online]. 2019, [cit. 2019-04-10]. Dostupné z: https://www.vive.com/eu/product/

[3] EPIC GAMES, Inc. *Unreal Engine Documentation* [online]. 2019, [cit. 2019-03-09]. Dostupné z: https://docs.unrealengine.com/en-us/

[4] LAGARDE, Sébastien; Charles de ROUSIERS (Electronic Arts Frostbite) *Moving Frostbite to Physically Based Rendering 3.0* [online]. 2014, [cit. 2019-03-06]. Dostupné z: https://seblagarde.files.wordpress.com/2015/07/course_notes_moving_frostbite_to_pbr_v32.pdf

[5] LAGARDE, Sébastien *PI or not to PI in game lighting equation* [online]. 2012, [cit. 2019-03-22]. Dostupné z: https://seblagarde.wordpress.com/2012/01/08/pi-or-not-to-pi-in-game-lighting-equation/

[6] Allegorithmic *THE PBR GUIDE - PART 1* [online]. 2019, [cit. 2019-03-06]. Dostupné z: https://academy.allegorithmic.com/courses/the-pbr-guide-part-1

[7] KARIS, Brian (Epic Games) *Real Shading in Unreal Engine 4* [online]. 2013, [cit. 2019-03-06]. Dostupné z: https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf

[8] EPIC GAMES, Inc. *Physically Based Materials* [online]. 2011, [cit. 2019-04-15]. Dostupné z: https://docs.unrealengine.com/en-us/Engine/Rendering/Materials/PhysicallyBased

[9] Allegorithmic *THE PBR GUIDE - PART 2* [online]. 2019, [cit. 2019-03-06]. Dostupné z: https://academy.allegorithmic.com/courses/the-pbr-guide-part-2

[10] LAGARDE, Sébastien *Feeding a physically based shading model* [online]. 2011, [cit. 2019-04-15]. Dostupné z: https://seblagarde.wordpress.com/2011/08/17/feeding-a-physical-based-lighting-mode/

[11] HOFFMAN, Naty *Physically Based Shading in Theory and Practice* [online]. 2016, [cit. 2019-03-06]. Dostupné z: https://blog.selfshadow.com/publications/s2015-shading-course//hoffman/s2015_pbs_physics_math_slides.pdf

[12] Luoshuang *Unreal Engine Documentation* [online]. 2018, [cit. 2019-04-09]. Dostupné z: https://forums.unrealengine.com/development-discussion/rendering/1460002-luoshuang-s-gpulightmass

[13] EPIC GAMES, Inc. *Lightmass Portals* [online]. 2019, [cit. 2019-03-10]. Dostupné z: https://docs.unrealengine.com/en-us/Engine/Rendering/LightingAndShadows/LightmassPortals

[14] EPIC GAMES, Inc. *Optimizing Your Game | Live Training | Unreal Engine Livestream* [online]. 2019, [cit. 2019-03-09]. Dostupné z: https://www.youtube.com/watch?v=U0p8EY07_mc

[15] NVIDIA *GeForce RTX: A Whole New Way To Experience* [online]. 2019, [cit. 2019-03-12]. Dostupné z: https://www.nvidia.com/en-us/geforce/news/graphics-reinvented-new-technologies-in-rtx-graphics-cards/#dlss

[16] HILL, Stephen *Counting Quads* [online]. 2012, [cit. 2019-03-25]. Dostupné z: https://blog.selfshadow.com/2012/11/12/counting-quads/

[17] EPIC GAMES, Inc. *Setting Up Automatic LOD Generation* [online]. 2019, [cit. 2019-04-10]. Dostupné z: https://docs.unrealengine.com/en-us/Engine/Content/Types/StaticMeshes/HowTo/AutomaticLODGeneration

[18] ERRIN, M. (Intel); ROUS, Jeff (Intel) *Unreal* Engine 4 Optimization Tutorial, Part 4* [online]. 2019, [cit. 2019-03-09]. Dostupné z: https://software.intel.com/en-us/articles/unreal-engine-4-optimization-tutorial-part-4

[19] Valve Corporation *Smoothing groups* [online]. 2018, [cit. 2019-04-15]. Dostupné z: https://developer.valvesoftware.com/wiki/Smoothing_groups

[20] Tech Art Aid. *UE4 Graphics Profiling: Pipeline and Bottlenecks* [online]. 2019, [cit. 2019-03-14]. Dostupné z: https://www.youtube.com/watch?v=UZH4vZ0NDAw&list=PLF8ktr3i-U4A7vuQ6TXPr3f-bhmy6xM3S&index=3

[21] TONKČI, Jukić *Draw calls in a nutshell* [online]. 2019, [cit. 2019-03-09]. Dostupné z: https://medium.com/@toncijukic/draw-calls-in-a-nutshell-597330a85381

[22] EPIC GAMES, Inc. *4.21 Visibility and Occlusion Culling* [online]. 2019, [cit. 2019-03-09]. Dostupné z: https://docs.unrealengine.com/en-us/Engine/Rendering/VisibilityCulling

[23] RYAN, Brucks. *Dither Temporal AA for Translucency - what am I doing wrong?* [online]. 2019, [cit. 2019-03-20]. Dostupné z: https://forums.unrealengine.com/development-discussion/rendering/106004-dither-temporal-aa-for-translucency-what-am-i-doing-wrong?p=906941#post906941

[24] fragmentbuffer *GPU Performance for Game Artists* [online]. 2017, [cit. 2019-04-15]. Dostupné z: http://fragmentbuffer.com/gpu-performance-for-game-artists/

[25] Tech Art Aid. *UE4 Optimization: Instancing* [online]. 2015, [cit. 2017-03-14]. Dostupné z: http://www.youtube.com/watch?v=oMIbV2rQO4k

[26] M., Marcus *Modular Concepts for Game and Virtual Reality Assets* [online]. 2019, [cit. 2019-03-15]. Dostupné z: https://software.intel.com/en-us/articles/modular-concepts-for-game-and-virtual-reality-assets

[27] LICHNOVSKÝ, Václav; MALÍNSKÝ, Jiří. *Přehled histologie člověka v obrazech 1* 2., nezměn. vyd. Ilustroval Zdeňka MICHALÍKOVÁ. Olomouc: Univerzita Palackého v Olomouci, 2009. ISBN 978-80-244-1769-1.

[28] LICHNOVSKÝ, Václav; MALÍNSKÝ, Jiří. *Přehled histologie člověka v obrazech 2* 2., nezměn. vyd. Ilustroval Zdeňka MICHALÍKOVÁ. Olomouc: Univerzita Palackého v Olomouci, 2009. ISBN 978-80-244-2277-0.

[29] EPIC GAMES, Inc. *Physics Constraint Component User Guide* [online]. 2019, [cit. 2019-04-09]. Dostupné z: https://docs.unrealengine.com/en-US/Engine/Physics/Constraints/ConstraintsBlueprints

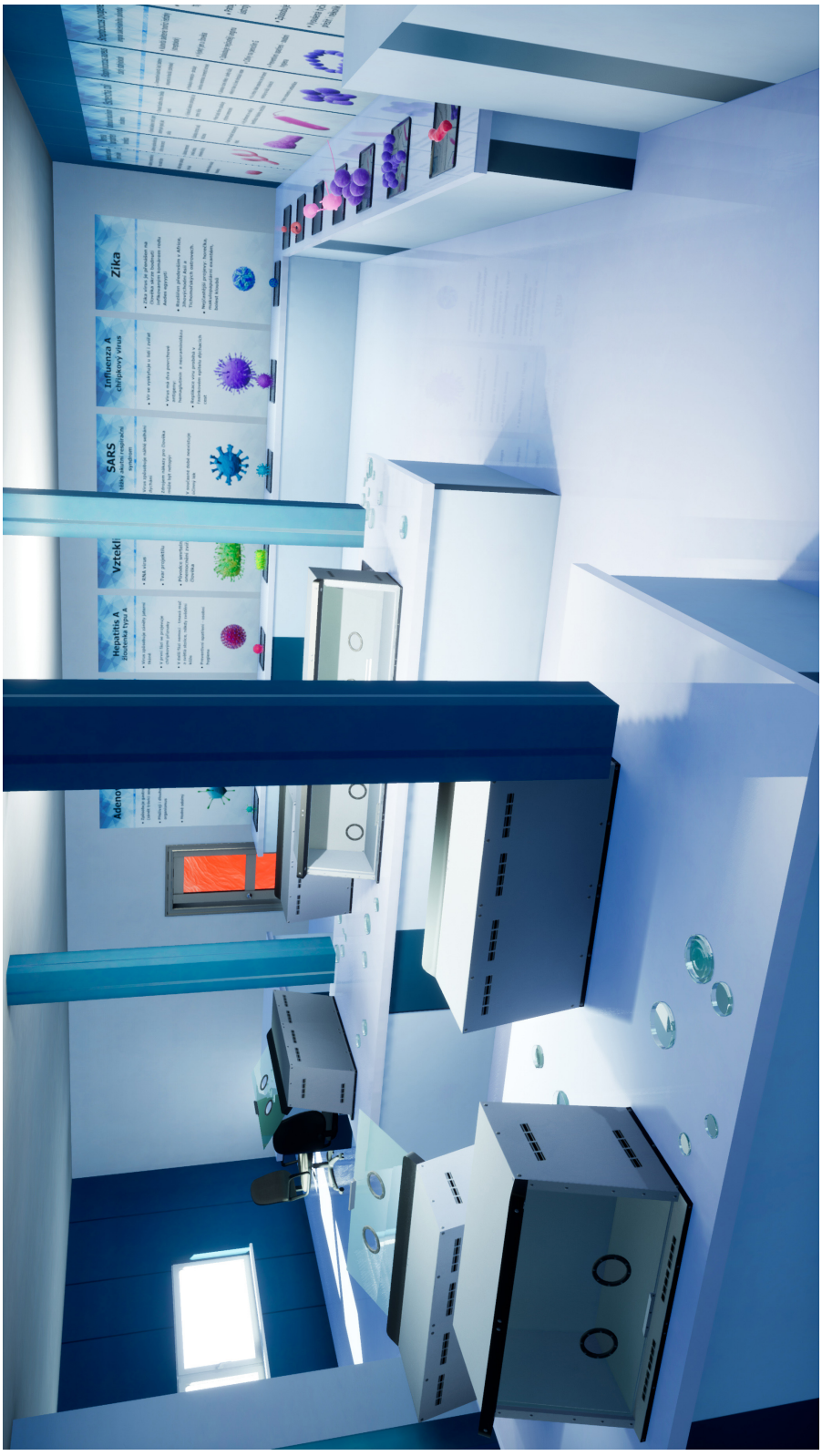
[30] EPIC GAMES, Inc. *APEX* [online]. 2019, [cit. 2019-04-12]. Dostupné z: https://docs.unrealengine.com/en-us/Engine/Physics/Apex

# Seznam Příloh

# A Ukázka struktury části souboru pro nastavení parametrů krevního řečiště

```
...

Level 3{
   FromScore = 50,
   SpawnRate = 1.2,
   Flow Speed = 1.3,
   RedBloodCell = 0.4,
   Lymphocyte = 0.2,
   LDLCholesterol = 0.1,

   AdenoVirus = 0.2,
   HepatitisVirus = 0.3,
   InfluenzaVirus = 0.4,
   RabiesVirus = 0.5,
   SARSVirus = 0.6,
   ZikaVirus = 0.6,
   EbolaVirus = 0.6,

   Streptococcus = 0.3,
   Staphylococcus = 0.3,
   Escherichia = 0.4,
   Bordetella = 0.3,
   Borrelia = 0.3,
   Tuberculosis = 0.4,
   Clostridium = 0.5,
}

Level 4{
   FromScore = 80,
   SpawnRate = 1.3,
   FlowSpeed = 1.4,
   RedBloodCell = 0.4,
   Lymphocyte = 0.2,
   LDLCholesterol = 0.1,

   AdenoVirus = 0.1,
   HepatitisVirus = 0.2,
   InfluenzaVirus = 0.3,
   RabiesVirus = 0.4,
   SARSVirus = 0.6,
   ZikaVirus = 0.8,
   EbolaVirus = 0.7,

   Streptococcus = 0.1,
   Staphylococcus = 0.2,
   Escherichia = 0.3,
   Bordetella = 0.3,
   Borrelia = 0.4,
   Tuberculosis = 0.5,
   Clostridium = 0.7,
}

...
```

# B Ukázka výsledné scény virtuální laboratoře



# C Ukázka výsledné scény krevního řečiště